L'Intelligence Artificielle

à travers Turbo Prolog

Bénédicte Hudault

ellipses

L'INTELLIGENCE ARTIFICIELLE

A TRAVERS TURBO PROLOG

par
BENEDICTE HUDAULT
Docteur ès sciences
en physique et en informatique
Maître de Conférence à l'Université de Paris I

ellipses

EDITION MARKETING
EDITEUR DES PREPARATIONS
GRANDES ECOLES MEDECINE
32, rue Bergue 75015 PARIS

ISBN 2-7298-9159-5

TABLE DES MATIÈRES

AVANT-PROPOS

Cet ouvrage est avant tout un ouvrage d'initiation qui s'éloigne volontairement de toute notic complexe. Son objet est de rendre les connaissances qui y sont développées accessibles à toi et non plus seulement à des spécialistes en intelligence artificielle.

Dans une première partie, le lecteur découvrira ce qu'est réellement l'intelligence artificiell cette application fascinante de l'informatique qui suscite angoisse, chez les uns, scepticism chez les autres. Puis, les immenses possibilités d'applications en ce domaine, bien qu'encor au stade de la recherche, sont évoquées. Et enfin, les principes de base de logique, sar lesquels il n'est pas possible d'appréhender Prolog, sont étudiés.

Il n'est pas nécessaire d'être informaticien pour aborder Prolog, nous serions même tentés d dire : au contraire ! Avec la deuxième partie de ce manuel , consacrée à l'apprentissage d Turbo Prolog, le lecteur découvrira une nouvelle logique de l'informatique : l "programmation logique" en opposition à la "programmation algorithmique".

S'initier à Prolog, c'est ouvrir une porte sur l'informatique future des langages naturels et de systèmes experts. C'est pourquoi, la troisième partie de cet ouvrage est consacrée à l réalisation effective de deux petits programmes d'intelligence artificielle en Turbo prolog : un système générateur de systèmes experts et un analyseur syntaxique.

PREMIÈRE PARTIE :

INTELLIGENCE ARTIFICIELLE ET SYSTÈME EXPERTS

1.1. QU'EST-CE-QUE L'INTELLIGENCE ARTIFICIELLE ?

Le but de l'intelligence artificielle est de simuler le cerveau humain en réalisant des "machines intelligentes". Il ne s'agit d'ailleurs en aucun cas de machines intelligentes mais bien plutôt de programmes intelligents.
L'intelligence présente de multiples facettes : nous sommes capables d'enrichir nos connaissances, de résoudre des problèmes, d'utiliser un langage, de mémoriser des informations etc. En un mot, qu'est-ce-que l'intelligence ? C'est une question qui préoccupe les philosophes depuis des siècles et laquelle nous ne tenterons pas de répondre.

1.1.1. Que faut-il entendre par "machines intelligentes" ?

La réponse nous est fournie par l'un des pères de l'informatique : Alan Turing. Celui-ci proposa en effet le scénario suivant pour déterminer si une machine pouvait être qualifiée d'intelligente : un homme dialogue par l'intermédiaire d'un clavier écran avec une entité distante. Si après une conversation prolongée celui-ci ne peut déterminer si son interlocuteur était un homme ou une machine alors, on peut attribuer à celle-ci l'étiquette "intelligente".
De telles expériences ont été réalisées. En particulier, le programme ELIZA (1965) simule les réponses d'un psychiatre durant une consultation avec un patient.

Exemple de dialogue :

```
Le client : Je suis déprimé la plupart du temps
ELIZA : Je suis navré d'apprendre que vous êtes déprimé.
Le client : C'est vrai, je suis malheureux
ELIZA  : Pensez-vous que le fait de venir ici va vous aider à
ne plus être malheureux ?
...
```

Le comportement d'ELISA semble très proche de celui de certains praticiens !.. Toutefois, si on examine attentivement un dialogue entre le programme et un client, on s'aperçoit rapidement que les réponses d'ELIZA ne sont en fait qu'une paraphrase sophistiquée des paroles du patient.
Le principe de base du programme est un certain nombre de phrases-clés, telles que :

- `Pensez-vous que XXX`
- `Parler-moi de XXX`
- `Je suis navré d'apprendre que XXX`

La séquence XXX est construite par le programme à partir de la dernière entrée du client. Par exemple, `"je suis déprimé"` sera transformé par le programme en `"vous êtes déprimé"` et réinséré dans la phrase `"Je suis navré d'apprendre que"`.

Dans le cas où le client ne répond que par monosyllabe, ELIZA est capable de contre-attaquer avec une phrase du genre :
"Vous ne semblez pas très coopératif aujourd'hui ?"
Le programme enfin est capable de choisir une phrase type en réduisant au minimum les répétitions.
ELIZA satisfait au test de Turing. On raconte même qu'un informaticien ayant tenté le test sur sa secrétaire, celle-ci l'aurait prié de quitter la pièce, la conversation prenant un tour trop intime ! Et pourtant, personne n'admet que le programme soit intelligent, puisqu'il ne comprend pas la conversation à laquelle il participe. Cette expérience nous donne à penser qu'à l'intelligence est attachée la compréhension.

1.1.2. Informatique traditionnelle-Intelligence artificielle

Résoudre un problème, (nous appelons problème, toute situation dont la résolution peut être confiée à un ordinateur) en informatique traditionnelle demande de concevoir un algorithme fournissant la solution.

1) Qu'est qu'un algorithme ?

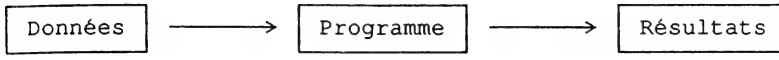
L'algorithme n'est pas une notion nouvelle due à l'informatique. En mathématiques, c'est une suite de calculs pour parvenir à la solution d'un problème.
Le plus vieil algorithme du monde est l'algorithme d'Euclide qui permet, par divisions successives de déterminer si deux nombres entiers sont premiers entre eux ou non.
En informatique, un algorithme est un suite d'actions élémentaires permettant de parvenir à la résolution d'un problème.
Mais il n'existe pas toujours un algorithme pour résoudre un problème : aucun algorithme au monde ne permet d'assurer la victoire dans une partie d'échecs, il serait pur folie d'essayer de concevoir un algorithme de traduction d'une langue dans une autre.
Le but de l'intelligence artificielle est précisément de tenter de résoudre les problèmes qui n'ont pas de solution algorithmique. Pour résoudre ce type de problèmes, on fait appel aux méthodes heuristiques (du grec ευρισ : qui a bon nez ! ou l'art de bien deviner !).

2) Qu'est-ce qu'une heuristique ?

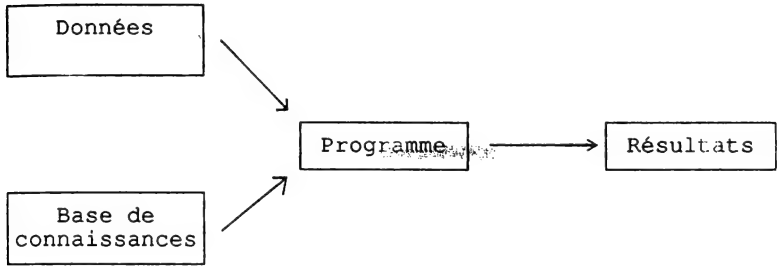
C'est une technique de résolution qui tient compte à chaque pas des résultats précédents et qui en déduit la stratégie à adopter par la suite. Les méthodes heuristiques s'opposent aux méthodes algorithmiques car elles n'assurent pas qu'une solution sera trouvée en un nombre fini de pas.
Les techniques heuristiques font usage de programmes d'apprentissage qui permettent de tenir compte des succès et échecs antérieurs.

Une autre différence entre l'informatique traditionnelle et l'intelligence artificielle réside dans la nature des objets manipulés. Alors que l'informatique traditionnelle manipule des données numériques ou alphanumériques un programme en intelligence artificielle doit manipuler des symboles (comme en algèbre) et des connaissances.
La réalisation enfin d'un programme d'intelligence artificielle nécessite qu'un expert humain procède à l'adaptation de ses connaissances.
En effet, en informatique traditionnelle, l'informaticien élabore un algorithme à partir de ses propres connaissances. Le programme reçoit des données et fournit un résultat.

En intelligence artificielle, la masse des connaissances est généralement trop volumineuse pour être assimilée par l'informaticien. L'expert formalise son savoir et le cogniticien traduit ces connaissances en une base de règles utilisable par un outil informatique.
Les connaissances sont décrites de manière déclarative de telle sorte qu'elles soient accessibles à un non-informaticien. Elles sont séparées du programme.



Programmation traditionnelle



Programmation en intelligence artificielle

1.2. APPLICATIONS DE L'INTELLIGENCE ARTIFICIELLE

Les champs d'application de l'intelligence artificielle s'étendent à tous les domaines où l'homme est amené à résoudre des problèmes. Alors que l'informatique traditionnelle exige fiabilité et exhaustivité des données numériques, l'intelligence artificielle est basée sur une approche des relations entre les variables et non sur les variables elles-mêmes. L'intelligence artificielle peut intégrer des informations incertaines et même l'absence d'informations. C'est pourquoi, les domaines d'application de l'intelligence artificielle semblent infinis.
Les systèmes experts, application la plus courante de l'intelligence artificielle, sont décrits dans le prochain chapitre.

1.2.1. Jeux de réflexion

Ils constituent un domaine de prédilection pour les chercheurs en intelligence artificielle. En effet, les jeux de stratégie (échecs, dames etc..) sont des problèmes bien définis : la configuration du terrain, les pièces, leurs déplacements sont parfaitement connus. Les règles sont rigoureuses et les résultats incontestables : une partie est gagnée, nulle ou perdue. Ils ne laissent aucune place au hasard.
Leurs structure enfin est arborescente : chaque coup correspond à un choix parmi un nombre fini de coups possibles et génère une nouvelle situation. Les jeux de réflexion constituent des structures hiérarchisées. Les méthodes utilisées sont généralisables.
On a réalisé des programmes jouant aux échecs, capables de battre très largement les joueurs moyens. Ils se situent à 2 200 points ELO (échelle utilisée pour le classement international

des joueurs) contre 2 400 pour les Grands Maîtres.
Le principe d'un programme de jeu est de développer une arborescence correspondant au plus grand nombre de déplacements possibles depuis la situation actuelle, tout en tenant compte des contraintes de temps et d'espace. L'arbre du jeu de dames contient près de 10^{40} noeuds et les échecs beaucoup plus encore !

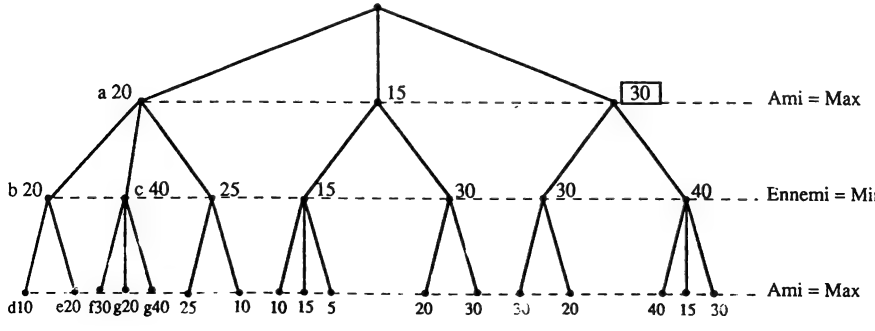
1) La fonction d'évaluation

Chaque feuille (élément terminal) de l'arbre est évalué à partir d'un fonction dite "d'évaluation", obligatoirement empirique (c'est à dire heuristique) et établie par le programmeur en fonction des règles du jeu. Aux échecs, par exemple, il faut tenir compte des menaces pouvant peser sur certaines pièces et au contrôle des cases stratégiques. Bien entendu, tout le programme va reposer sur cette fonction d'évaluation. Une fonction d'évaluation contient une certaine connaissance du jeu mais jamais toutes les connaissances. Sauf bien évidemment, dans les jeux où un algorithme assure la victoire quand on a joué le premier (par exemple le jeu de Nim, le drapeau anglais,...) mais ces jeux ne sont pas de notre propos.

2) Choisir un coup ou minimax

Le minimax est une méthode pour choisir le coup le plus avantageux. Sa logique repose sur l'hypothèse selon laquelle le joueur choisira toujours le coup correspondant à la fonction d'évaluation maximum alors que l'adversaire lui, choisira toujours le coup correspondant à la fonction d'évaluation minimum.
Le programme va donc :
(1) Générer l'arborescence jusqu'à une profondeur "N".
(2) Evaluer les feuilles.
(3) Affecter au noeud de niveau immédiatement supérieur la valeur maximum des feuilles auquel il a donné naissance. Ceci pour un coup ami. Pour un coup ennemi, c'est la valeur minimum des feuilles qui sera remontée. Les noeuds de niveau "N-1" deviennent les nouvelles feuilles.
(4) Recommencer l'étape (3) jusqu'à ce que l'on atteigne le niveau "1".
(5) Jouer le coup correspondant à la valeur maximale.

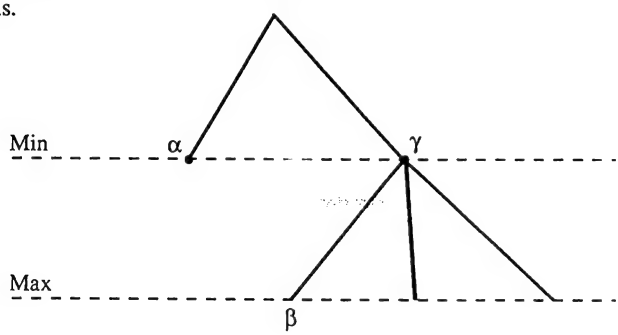
Exemple de minimax sur un arbre de profondeur 3



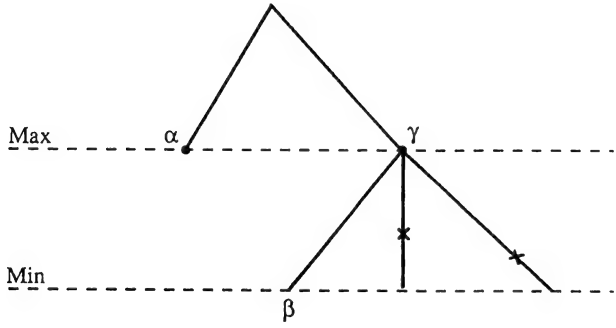
Le coup joué sera celui encadré.

3) Elaguer l'arbre de recherche ou l'α - β

L'α - β consiste à éliminer de l'arbre toutes les branches inutiles.
Observez l'arbre donné ci-dessus. A profondeur 2, le choix correspond à un minimum, or le noeud *b* a la valeur 20 et le noeud *f* (profondeur 3) à la valeur 30. On sait que la valeur qui sera affectée au noeud *c* sera supérieure ou égale à 30. Quelque soit la valeur des feuilles *g* et *h*, le programme affectera la valeur 20 au noeud *a*. Il est donc inutile d'évaluer les feuilles *g* et *h*. Le processus se généralise à tous les niveaux de l'arbre pour les coups amis et les coups ennemis.



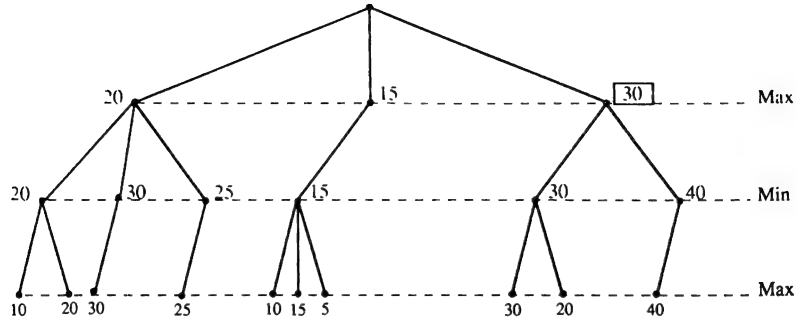
Si β > α les autres branches issues de γ peuvent être ignorées.



Si β < α les autres branches issues de γ peuvent être ignorées.

L'α - ß est une technique efficace pour réduire l'arbre de recherche. Pour un arbre à trois branches d'une profondeur quatre, dans le cas le plus favorable, celui-ci ne contiendra que 21 noeuds au lieu de 81.

Exemple : effet de l'α - β sur l'arbre donné ci-dessus



4) L'apprentissage

Les joueurs humains améliorent continuellement leurs jeux par l'expérience. C'est pourquoi, on crée des programmes dits d'apprentissage capables de modifier leur fonction d'évaluation. Le problème est de savoir comment modifier la fonction d'évaluation. Il existe plusieurs méthodes d'apprentissage : direct, par mémorisation, par conseil...
Le principe de l'apprentissage direct consiste à évaluer chaque noeud de deux manières différentes : d'un part par le minimax et d'autre part par un calcul direct de la fonction d'évaluation, puis de comparer les deux valeurs obtenues. Supposons que la fonction d'évaluation soit de la forme :

$$F = p_1f_1 + p_2f_2 + .. + p_nf_n$$

où les f_i sont les facteurs du jeu et les p_i des poids variables par l'intermédiaire desquels on pourra modifier la fonction d'évaluation. Si les deux valeurs obtenues par minimax et par calcul direct sont voisines, la fonction d'évaluation est satisfaisante. Si elles sont très différentes, on modifie les poids affectés à chaque facteur de manière à ajuster la fonction d'évaluation.

Dans l'apprentissage par mémorisation, le programme mémorise chaque situation et son évaluation. Lorsque la même situation est rencontrée, la valeur est retrouvée.

Dans l'apprentissage par conseils, l'expert peut fournir des informations au logiciel. Cette aide est généralement transmise au programme sous forme de règles. Ce module peut s'avérer très complexe si les conseils de l'expert sont donnés en langage naturel.

1.2.2. Compréhension du langage naturel

Les premiers systèmes créés pour le traitement du langage naturel avaient pour but la traduction automatique. Ils utilisaient essentiellement des dictionnaires et l'analyse syntaxique (grammaticale). Or, il apparut très rapidement que la seule analyse grammaticale n'était pas suffisante pour réaliser une traduction correcte. Celle-ci passe nécessairement par la compréhension du texte : il n'est pas possible d'ignorer la signification des mots ou encore leur sémantique.

Il convient donc :

a) De résoudre les ambiguïtés sémantiques des mots comme dans la phrase suivante :
La petite ferme la cache.
Peut-être ne remarquez-vous pas la double signification de cette phrase. Alors que dans les phrases :
Catherine essaye de voir la rivière, mais la petite ferme la cache.
et
Avant de quitter la pièce, la petite ferme la cache.
L'ambiguïté est parfaitement levée. Ce qui prouve que la compréhension de phrases comprend un grand nombre de mécanismes.

b) De résoudre l'ambiguïté de la référence des pronoms, comme dans la phrase suivante :
La voiture de ma grand-mère qui cliquette.
Grammaticalement le pronom *"qui"* se rapporte à *"grand-mère"* !
Une manière d'aborder le problème de la compréhension du langage naturel est de limiter le texte à un domaine *très restreint* et de donner au programme des connaissances sur le sujet par l'intermédiaire de "scripts" (R. Schank).
Un script est un mot-clé qui permet d'inférer d'autres événements. Ces événements sont, bien entendus, susceptibles d'être rencontrés dans le texte.
L'exemple donné par Shank est celui du restaurant. Dès que le programme a rencontré ce mot.
- Il connaît les acteurs : le client, le cuisinier , le garçon, le maître d'hôtel etc.
- Il connaît également les objets manipulés : les tables, le menu, la carte, l'argent etc.

Les applications pratiques de la compréhension du langage naturel sont innombrables : traduction automatique, sommaires automatiques, réalisation d'interfaces homme-machine.

1.2.3. Reconnaissance des formes

La reconnaissance des formes est un domaine de l'intelligence artificielle où l'on désire donner aux ordinateurs la possibilité d'interpréter et d'analyser des représentations visuelles. C'est un secteur riche en applications pratiques et intéressantes, par exemple :
- La lecture directe par les ordinateurs de textes imprimés ou manuscrits.
- Donner aux robots de l'industrie des possibilités visuelles et tactiles.
- L'analyse informatique de photographies, que ce soit en matière de météorologie, de surveillance militaire ou de protection de l'environnement.

On retrouve en matière de reconnaissance des formes, les mêmes problèmes que ceux rencontrés en compréhension du langage naturel, à savoir le rôle que joue la connaissance de l'environnement.
L'exemple simple suivant montre que la connaissance joue un rôle important dans la reconnaissance visuelle.
Le symbole + est perçu de deux manières différentes dans les contextes suivants :

$$c = a + b$$

$$l\,a\,+\,a\,b\,l\,e$$

La connaissance utilisée est ici particulièrement simple pour ne pas dire simpliste : le symbole + n'est pas une lettre de l'alphabet.
Les programmes de perception automatique opèrent dans des domaines très restreints (comme pour la compréhension du langage naturel). L'analyse de scènes quelconques du monde réel relève plus de la science fiction aujourd'hui que de l'intelligence artificielle.

On peut, en simplifiant, résumer les étapes de la reconnaissance visuelle, aux processus suivants :
(1) Création d'une matrice d'intensité, c'est à dire réalisation d'une fine grille de carrés, chacun contenant un nombre représentant l'intensité lumineuse dans la zone correspondante de l'image.
(2) Analyse de la matrice d'intensité pour mettre en évidence les bords et les angles de chaque objet.
(3) Applications d'heuristiques pour transformer les ébauches obtenues à l'étape (2) en objets complets.
(4) Identification des objets obtenus en (3) avec les objets autorisés par le système.
Bien entendu, les étapes décrites ci-dessus ne sont pas menées indépendamment les unes des autres mais peuvent être fortement imbriquées.

1.3. SYSTEMES EXPERTS

1.3.1. Qu'est ce qu'un système expert ?

Un système expert est un ensemble de programmes capables de simuler la démarche d'un expert humain à la recherche d'un diagnostique dans son domaine de compétence.
La valeur d'un expert humain réside dans la connaissance qu'il a de son domaine et également dans sa capacité de l'exploiter judicieusement, il en est de même pour un système expert.
Un système expert est un intermédiaire entre l'expert humain qui transmet ses connaissances au système et l'utilisateur humain qui a recours au système expert à la fois pour résoudre ses problèmes et pour enrichir ses connaissances sur le domaine.
Le but d'un système expert est de mettre à la disposition de l'utilisateur le mécanisme intellectuel d'un spécialiste d'un domaine particulier par l'intermédiaire d'un ordinateur.
Un système expert aura l'avantage sur un expert humain d'être toujours disponible et duplicable par simple copie du logiciel.
Les systèmes experts constituent, en quelque sorte, une synthèse de toutes les applications de l'intelligence artificielle. En effet, tout système expert doit être capable de dialoguer en langage naturel, de raisonner, d'expliquer son raisonnement, d'acquérir des connaissances nouvelles, de tenir compte de ses échecs et succès antérieurs. Il doit être capable enfin de tenir compte d'un certain flou dans les connaissances (coefficient de vraisemblance).

1.3.2. Quelques systèmes experts

Avant d'étudier le principe des systèmes experts, nous dirons un mot des systèmes experts les plus connus et réellement opérationnels à ce jour.
La première tentative de système expert fut DENDRAL, programme américain d'analyse chimique à partir de spectrogrammes, qui date des années 70.
La première véritable réalisation d'un système expert est MYCIN en 1976. MYCIN est un programme de diagnostique et de traitement des maladies infectieuses du sang. C'est le plus connu des systèmes experts. MYCIN dialogue en langage naturel. Dans une première partie, le dialogue est mené par le système afin de recueillir les informations nécessaires sur le patient, c'est à dire établir la base de faits. Une fois la base de faits établie, MYCIN établit son diagnostic en utilisant une base de deux cents règles environ. Noter qu'à chaque fait est associé un coefficient de vraisemblance (de -1 à +1) permettant un raisonnement incertain : -1 le fait est faux, +1 le fait est certain.

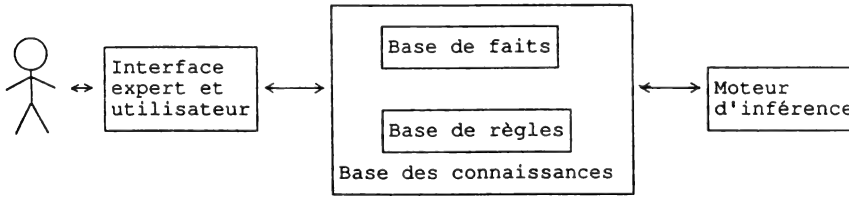
Un autre système expert très célèbre est PROSPECTOR (1979). Son objectif est d'aider les géologues dans leur diagnostic sur la composition en minerais d'une zone minière. PROSPECTOR a fait en 1980 une prédiction sur un gisement de molybdène dans l'état de Washington qui a été confirmé ultérieurement par forage.

Tous les systèmes experts opérationnels aujourd'hui sont postérieurs aux années 76. Ils touchent tous les domaines d'application :
- La médecine qui est certainement le terrain rêvé pour de tels systèmes, les informations étant le plus souvent quantifiables.
- La chimie
- La physique
- La géologie
- La botanique
- Les mathématiques
- Le droit
- Le bâtiment

et bien d'autres encore..
Signalons également, dans de nombreux domaines, l'existence de systèmes d'aide d'urgence sur les décisions à prendre en cas de panne.

1.3.3. Structure d'un système expert

Un système expert est composé de trois modules :
- la base de connaissances
- le moteur d'inférence
- les interfaces



Systèmes experts

La base de connaissances, donnée sous forme déclarative, contient toutes les compétences spécifiques au domaine de l'expertise. Elle comprend deux parties :
- la base de faits qui contient les faits connus et les définitions.
- la base de règles qui permet à partir des faits connus d'établir des faits nouveaux.

Ces règles se présentent sous la forme : SI prémisses ALORS conclusion

Le moteur d'inférence est la partie centrale du système expert, c'est un logiciel qui est chargé d'exploiter les connaissances et d'effectuer les déductions. C'est lui qui donne l'impression que le système expert raisonne. Inférer est une opération intellectuelle par laquelle on passe d'une vérité à une autre, jugée vraie en raison de son lien avec la précédente.

Les interfaces expert et utilisateur sont conçues de manière à rendre le système expert convivial. Elles utilisent des dictionnaires où figurent tous les mots et symboles susceptibles d'être utilisés par l'expert ou l'utilisateur.

Moteur d'inférence et interfaces sont indépendants du domaine de l'expertise, et constituent ce que l'on appelle le noyau du système expert.

1.3.4. Comment réalise-t-on un système expert ?

La réalisation d'un système expert impliquera toujours une codification du savoir (création de la base des connaissances), ce qui nécessite deux compétences : l'expert humain et le cogniticien.
La présence d'un expert humain est bien entendu indispensable. Qui songerait à réaliser un système expert juridique sans juriste ? Le rôle du cogniticien sera d'aider l'expert à formaliser son savoir.
La réalisation du moteur d'inférence et des interfaces dépend beaucoup de l'outil informatique utilisé. Elle sera étudiée dans un prochain chapitre.

1.4. NOTIONS DE LOGIQUE

1.4.1. Logique des propositions

Une proposition logique est une assertion qui ne peut prendre qu'une des deux valeurs VRAI ou FAUX.
Par exemple, l'assertion : "3 est supérieur à 2" est une proposition logique ayant la valeur VRAI.
Une fois définies les propositions logiques, nous pouvons en définir d'autres à l'aide des opérateurs logiques (NON, ET, OU, l'implication => et l'équivalence <=>). définis par leur table de vérité (P et Q désignent des propositions logiques) :

NON

P	NON P
VRAI	FAUX
FAUX	VRAI

Le NON logique correspond à la négation utilisée en langage naturel.
Par exemple, NON "3 supérieur à 2" est faux

ET

P	Q	P ET Q
FAUX	FAUX	FAUX
FAUX	VRAI	FAUX
VRAI	FAUX	FAUX
VRAI	VRAI	VRAI

P ET Q est vraie si P et Q sont simultanément vraies.
P ET Q est fausse si l'une au moins des deux propositions P ou Q est fausse.

OU

P	Q	P OU Q
FAUX	FAUX	FAUX
FAUX	VRAI	VRAI
VRAI	FAUX	VRAI
VRAI	VRAI	VRAI

P OU Q est vraie si l'une au moins des propositions P ou Q est vraie.
P OU Q est fausse si P et Q sont simultanément fausses.

L'implication =>

P	Q	P => Q
FAUX	FAUX	VRAI
FAUX	VRAI	VRAI
VRAI	FAUX	FAUX
VRAI	VRAI	VRAI

L'implication correspond à l'implication utilisée dans le langage courant :
P => Q est vraie si P et Q sont simultanément vraies ou si P est fausse.
P => Q est fausse si P est vraie et Q fausse.
Par exemple, l'assertion logique : "Si Jean est Français, Jean est Européen" est une implication logique entre les deux propositions P et Q :
P = Jean est Français
Q = Jean est Européen.
Si Jean est Français alors Jean est Européen. Les deux propositions P et Q sont vraies simultanément.
En revanche, si Jean n'est pas Français (P fausse), Q peut être vraie ou fausse (Jean peut être Européen ou non).
La règle du Modus Ponens, à la base de tous les processus démonstratifs mis en oeuvre par un moteur d'inférence, traduit la dernière ligne de la table de vérité :
Si P => Q est vraie et si P est vraie alors on peut en déduire que Q est vraie.

L'équivalence <=>

P	Q	P <=> Q
FAUX	FAUX	VRAI
FAUX	VRAI	FAUX
VRAI	FAUX	FAUX
VRAI	VRAI	VRAI

P <=> Q est vraie si P et Q ont même valeur de vérité.
Par exemple l'assertion : "Jean et Catherine sont mariés" traduit une équivalence entre les deux propositions P et Q :
P = Jean est le mari de Catherine
Q = Catherine est la femme de Jean
Ces deux propositions ont même valeur de vérité, elles traduisent une double implication entre P et Q.

Il ne faut pas faire de confusion entre l'implication et l'équivalence. Si P => Q est vraie, lorsque P est fausse, Q peut être vraie ou fausse. Alors que si P <=> Q est vraie, si P est fausse, Q est fausse également.

1.4.2. Calcul des prédicats

Si nous voulons exprimer en logique des propositions le fait que : "Tous les Français sont des Européens". Nous serons obligés d'énumérer toutes les propositions :
"Si François est Français alors François est Européen"
"Si Dominique est Français alors Dominique est Européen"

La logique des propositions ne permet pas de manipuler des classes d'objets. Elle ne possède pas la notion de variables. Un prédicat est une assertion ne pouvant prendre que les deux valeurs VRAI ou FAUX mais dont la valeur dépend de ses arguments.
La notion de prédicat utilise la notion de variables. Une fois les variables substituées à des objets réels, le prédicat devient une proposition logique.
Par exemple, pour exprimer que : "Tous les Français sont des Européens", nous écrirons en calcul des prédicats :
"Si X est Français alors X est Européen"
X est une variable, à laquelle on pourra substituer tous les Français.
Cette opération de substitution des variables par des objets réels est un des mécanisme de base des moteurs d'inférence et s'appelle *l'unification.*

1.4.3. Règles de production

Une règle de production est une implication logique du type P => Q, dans laquelle P désigne les prémisses (P = P1 ET P2 ..ET Pn) et Q la conclusion, que l'on écrira sous la forme :
SI P ALORS Q
ou encore : Q SI P

A chaque fois que P est vraie on peut en déduire que Q est vraie par la règle du Modus Ponens.
Par exemple, l'implication logique "S'il y a une épidémie de grippe et si vous avez mal à la tête et si vous avez des courbatures et si vous avez de la température alors vous avez la grippe", est un règle de production.
Avec les prémisses :
P1 = il y a une épidémie de grippe
P2 = vous avez mal à la tête
P3 = vous avez des courbatures
P4 = vous avez de la température
et la conclusion :
Q = vous avez la grippe
A chaque fois que les prémisses P1, P2, P3 et P4 sont vrais, on peut en déduire que la conclusion Q est vraie.

1.5. MOTEURS D'INFERENCE

Un moteur d'inférence est un programme capable de :
- lire et coder les faits et les règles de production.
- mettre en oeuvre ces règles.

1.5.1. Moteur d'ordre zéro - Moteur d'ordre un

Si les règles sont écrites en logique des propositions, on a un moteur d'ordre zéro.
Si les règles sont écrites en calcul des prédicats, on a un moteur d'ordre un.

1.5.2. Fonctionnement d'un moteur

Le moteur d'inférence peut être activé par trois modes de raisonnement : le chaînage avant, le chaînage arrière ou le chaînage mixte.

1) Chaînage avant
Il correspond au raisonnement déductif : que puis-je faire des informations disponibles ? Le moteur part des faits établis et utilise les règles applicables. On aboutit à d'autres faits qui permettent de déclencher de nouvelles règles. Le moteur fonctionne par saturation. Il n'est pas nécessaire de fixer un but pour le faire fonctionner. Il existe trois causes d'arrêt :
- toutes les règles ont été déclenchées sans résultat.
- il n'y a plus de règles applicables.
- on a obtenu le résultat désiré.

2) Chaînage arrière
Encore appelé remontée, retour-arrière ou back-track, le chaînage arrière correspond au raisonnement inductif. Le moteur part du but à démontrer et recherche une règle ayant le but comme conclusion. Si les prémisses de la règle appartiennent à la base de faits alors le but est vérifié. Si l'un des prémisses n'appartient pas à la base de faits, il devient un nouveau but, et le moteur part à la recherche de nouvelles règles applicables, et ainsi de suite.

3) Chaînage mixte

Il synthétise les deux modes de chaînage avant et arrière. Le moteur procède quasi simultanément en induction et en déduction.

Les moteurs d'inférence performants doivent pouvoir raisonner en chaînage mixte. En effet, si on part d'un certain nombre de faits et que l'on ne connaît pas le but à atteindre, le chaînage avant est l'approche naturelle. En revanche, si on a un ou plusieurs buts à atteindre, il est intéressant d'utiliser le chaînage arrière.

Par ailleurs, on peut considérer que nous gérons deux types de connaissances : les connaissances certaines et les connaissances empiriques ou incertaines.

Les connaissances incertaines sont utilisées en chaînage avant : elles évoquent des buts possibles. Les connaissances certaines sont ensuite utilisées en chaînage arrière pour confirmer ou infirmer les buts précédemment évoqués.

1.5.3. Acquérir un moteur d'inférence

1) Prolog

Le langage Prolog est un moteur d'inférence d'ordre un, fonctionnant en chaînage arrière. Il est donc particulièrement adapté à la réalisation de nouveaux moteurs d'inférence (fonctionnant en chaînage mixte par exemple), de systèmes experts ou de P.G.S.E.

2) Programmes Générateurs de Systèmes Experts

On trouve aujourd'hui sur le marché des noyaux (moteur d'inférence et interfaces) vendus sous le nom de Programmes Générateurs de Systèmes Experts (P.G.S.E.).

Citons parmi les plus courants pour compatibles PC : VP-Expert, Mimi, Esie..

Les plus puissants parmi les PGSE sont capables d'induire une base de connaissances à partir d'une base de données créée par un Système de Gestion de Base de Données (S.G.B.D.).

1.5.4. Réaliser un moteur d'inférence

Avec un langage traditionnel comme Pascal ou C, la réalisation d'un moteur d'inférence d'ordre un est une tâche délicate mais possible.

En revanche, s'il est assez facile de réaliser un moteur d'inférence d'ordre zéro en chaînage avant ou arrière en Basic, la pauvreté de ce langage ne permet pas de réaliser un moteur d'ordre un.

Bien entendu, les langages de l'intelligence artificielle : Prolog, Lisp et Logo sont beaucoup plus adaptés à la réalisation de moteurs d'inférence.

1.6. LANGAGES DE L'INTELLIGENCE ARTIFICIELLE

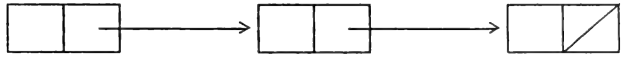
Les langages de l'intelligence artificielle sont au nombre de trois : Lisp, Logo et Prolog. Le but de ce chapitre est de présenter rapidement au lecteur les grandes caractéristiques de chacun de ces langages afin de mettre en évidence leurs similitudes et leurs différences.

Enfin, dans un dernier paragraphe, nous dirons un mot de la programmation orientée objets. Le problème étant de savoir si ce mode de programmation est adaptée à la réalisation d'applications en intelligence artificielle.

1.6.1. Similitude entre les différents langages de l'intelligence artificielle

- En Lisp aussi bien qu'en Logo ou en Prolog, la notion d'instructions n'existe pratiquement plus.
- La logique de ces langages ne fait référence qu'à la notion de vrai/faux : tout ce qui n'est pas faux est réputé vrai et réciproquement.
- La programmation est déclarative.
Au contraire des langages classiques comme Basic, Pascal ou C, pour lesquels la programmation est dite impérative (on décrit les étapes à exécuter), le traitement consiste non pas tant à modifier la valeur d'une variable qu'à produire un résultat par l'intermédiaire de fonctions (les prédicats).
- La notion de données structurées se limite aux listes. Il n'existe ni tableaux ni enregistrements dans ces langages.

Remarque :
Une liste est une structure de données chaînées, c'est à dire que chaque élément possède un lien (un pointeur) avec l'élément suivant. On ne peut parcourir une liste que de manière séquentielle.



- La récursivité (c'est à dire la possibilité pour une fonction de se rappeler elle-même) sera la technique de programmation la plus utilisée. En effet, seule la récursivité permet de trouver un élément dans une liste en Prolog où les structures de boucles compteur ou conditionnelles n'existent pas. Si ces mêmes structures existent en Lisp et en Logo, il est peu recommandé de les utiliser. Elles sont contraires à l'esprit du langage, un peu comme le GOTO du Pascal dont il ne faut jamais faire usage.
- Ce sont des langages symboliques.
Alors que les langages traditionnelles manipulent des données numériques ou alphanumériques, ces langages manipulent des symboles comme l'algèbre.

1.6.2. Lisp

Le langage Lisp (de l'Anglais "List Processing", alias : "Traitement de listes") a été créé vers les années 60 par un chercheur américain, ce qui est remarquablement précoce si l'on songe que le premier compilateur Fortran est apparu vers les années 54. Il fut très vite adopté par les chercheurs en intelligence artificielle : on a dit que Lisp était l'assembleur des langages de l'intelligence artificielle.
Il est certain qu'un grand nombre de programmes de cette branche de l'informatique n'auraient pas vu le jour sans l'existence de ce langage, cependant Lisp obéit encore à une logique algorithmique. Conçu à une époque, où la seule approche informatique était l'analyse algorithmique, la réalisation de logiciels en Lisp veut que la solution du problème soit trouvée manuellement par l'informaticien avant toute programmation.
L'une des grandes particularités de Lisp est de ne pas faire de distinction entre le programme et les données. Lisp manipule des symboles sous forme d'atomes et de listes et un programme en Lisp est une liste.

- Un atome est le plus petit élément manipulable du langage.
- Une liste est composée d'atomes ou de listes, elles-mêmes composées d'atomes ou de listes, elles mêmes composées..

Du fait de l'homogénéité de l'information mémorisée en Lisp, il est possible d'exécuter des données et de manipuler des fonctions. C'est cette propriété qui fait la puissance de ce langage. En effet, en intelligence artificielle, il n'existe pas de séparation nette entre le raisonnement et l'information, mais des relations entre divers concepts d'une part et des méthodes pour accomplir des tâches d'autre part.
En lisp, il est possible d'effectuer des traitements sur des expressions mathématiques, sans devoir les calculer explicitement. On peut travailler sur un polynôme notamment, sans se soucier de la valeur des constantes et des variables. Ce qui ne signifie pas pour autant que Lisp ne soit pas capable de traiter une expression au niveau arithmétique (retourner sa valeur).
Avec Lisp, il est possible de manipuler des expressions symboliques, ce qui est impossible avec les langages de programmation traditionnels qui établissent une séparation absolue entre le code et les données. On peut dire que Lisp est aux langages traditionnels ce que l'algèbre est à l'arithmétique.

Le fait de ne pas faire de distinction entre code et données n'est pas la seule raison qui donne à Lisp la possibilité de manipuler des symboles, cela tient également à la structure des atomes.
En effet, un atome est une sorte de "record" (d'enregistrement), formé de quatre champs, contenant respectivement :
- la valeur de l'atome en tant que variable.
- la définition de l'atome comme une fonction.
- le nom de l'atome.
- la liste des propriétés attachées à l'atome.

Ce sont les trois premiers champs de l'atome qui permettent de le manipuler comme une valeur, une fonction ou un symbole.
La liste des propriétés attachées à un atome permet, quant à elle, de créer très facilement des bases de données.

Lisp (quelque-soit la version du langage utilisé) est un langage interprété donc interactif, au contraire de Prolog qui existe en version interprétée (D-Prolog) ou compilée (Turbo Prolog).

L'algorithme général de l'exécution du Lisp est une boucle sans fin au cours de laquelle, les expressions lues au clavier sont évaluées et leurs résultats affichés.
C'est un langage fonctionnel (Lisp ne connaît pas les procédures).
La syntaxe du langage, n'en déplaise aux inconditionnels de Lisp, est fort pénible. Lisp utilise la notation polonaise préfixée (les opérateurs figurent devant les arguments, ce qui est déjà une démarche inhabituelle) et l'usage immodéré des parenthèses rebute les programmeurs les plus chevronnés (le problème du parenthèsage correct est l'un des plus irritant qui soit !)..
L'expression arithmétique `(1 + 2) * (3 + 4)` par exemple, s'écrira en Lisp :

```
(* (+ 1 2) (+ 3 4))
```

1.6.3. Logo

Logo est un langage méconnu, conçu pour les enfants : il n'est pas pour les adultes ! En réalité, Logo est un langage très puissant qui ne consiste pas uniquement, comme le croit le

plus grand nombre, à déplacer une tortue sur l'écran même, si c'est cet aspect de Logo qui a fait sa réputation.
Logo est en réalité un sous ensemble, on peut dire un dialecte de Lisp dont il reprend tous les principes :
- Langage symbolique.
- Pas de distinction entre le code et les données.
- Traitement des listes.
- Récursivité.
- Structure d'enregistrement de l'atome avec ses quatre champs (valeur, fonction, nom, liste de propriétés).

La syntaxe de Logo est infiniment plus agréable que celle de Lisp !
Lorsque l'on sait que l'on peut utiliser Lisp en ne connaissant qu'une douzaine des fonctions élémentaires (des primitives) du langage. On regrette vivement que l'initiation n'ait pas lieu à partir de Logo qui contient bien plus de douze primitives (donc bien plus qu'il n'en faut) et qui permet d'aborder tous les concepts passionnants mis en oeuvre par Lisp, d'une manière autrement moins rebutante.
La convivialité de Logo, bien qu'imparfaite est nettement supérieure à celle de Lisp. Les messages d'erreurs, en particulier, ont le mérite d'être compréhensibles, ce qui n'est pas le cas en Lisp où l'on a malheureusement le sentiment que l'on a cherché à les rendre le plus inintelligibles possibles !

1.6.4. Prolog

Le langage Prolog (de "Programmation Logique", en opposition à Programmation Algorithmique) est le langage né de l'intelligence artificielle. Il a été créé vers les années 75 à l'université d'Aix-Marseille. Avec Prolog est née une nouvelle ère de l'informatique : le passage de la programmation algorithmique à la programmation logique.
Prolog a été adopté par le projet japonais des ordinateurs de la cinquième génération.
Avec Prolog, c'est une autre logique informatique qu'il faut acquérir, c'est pourquoi aborder ce langage est souvent plus accessible à un débutant, qu'à un informaticien rompu au raisonnement algorithmique.
En Prolog, la notion d'instructions n'existe absolument plus. Vous n'y trouverez plus également, ni étiquettes, ni boucles, ni alternatives, ni aiguillages multiples, ni procédures, ni pointeurs. Programmer en Prolog, c'est créer une définition de son problème qui comporte en elle-même sa propre recherche de la solution : Prolog est un langage déclaratif ou encore un langage de représentation des connaissances.
Prolog met l'accent sur les relations, en permettant non plus de compter, de trier, de programmer des données mais au contraire, d'énoncer des connaissances, de définir des problèmes et de déduire des solutions.
Les logiciels écrits en Prolog ne sont jamais figés. En effet, il est possible d'augmenter, de modifier les performances de son application en modifiant sa base de connaissances et ceci sans toucher aux programmes. C'est le moteur d'inférence (c'est à dire Prolog) qui est chargé de prendre en compte ces nouvelles connaissances et de les rattacher aux connaissances antérieures. Chaque nouvelle information est énoncée et rajoutée indépendamment des autres.
Aucun langage de programmation enfin, ne possède une syntaxe aussi simple que Prolog. L'identité syntaxique entre données et programmes permet une manipulation simple du logiciel.

1.6.5. Programmation orientée objets

Le principe fondamental de la programmation orientée objets est d'associer le code et les données pour créer un ensemble (un objet). La programmation orientée objets introduit des concepts fondamentalement différents de la programmation traditionnelle. Il faut raisonner en termes de boîtes de "savoir-faire" et non plus en termes de procédures recevant des paramètres. Si ces techniques sont adaptées aux problèmes de simulation, il ne nous semble pas utile en intelligence artificielle de mettre l'accent sur les objets mais bien plutôt sur les relations.

1.6.6. Conclusion

Mise à part les langages orientés objets qui, nous l'avons dit, semblent peu adaptés à l'intelligence artificielle, les autres langages au contraire, se complètent.
Il n'est pas rare en effet, que les utilisateurs de Lisp rajoutent à ce dernier, toutes les fonctions qui lui manquent et que l'on trouve dans Prolog. C'est à dire essentiellement, un moteur d'inférence fonctionnant en chaînage arrière et capable d'utiliser l'unification (c'est à dire substituer à des variables des objets réels).
Néanmoins, Prolog a fait faire un grand pas en avant à l'intelligence artificielle. C'est un langage tout à fait original par rapport à tous les langages existants. Il est en outre beaucoup plus facile à aborder et à maîtriser que Lisp. La simplicité de sa syntaxe permet, sans contestation possible, une mise au point beaucoup plus rapide et aisée des programmes. Quand à la convivialité de Turbo Prolog, c'est un logiciel signé Borland, c'est tout dire !

DEUXIÈME PARTIE :

TURBO PROLOG

2.1. PRESENTATION GENERALE DU LOGICIEL

2.1.1. Configuration minimale requise

Pour utiliser Turbo Prolog, vous devez disposer :
- d'un IBM PC ou compatible.
- d'au minimum 384 Ko de mémoire vive.
- du DOS 2.0 ou ultérieur.

2.1.2. Contenu des quatre disquettes

Disquette 1/4

README.*	Fichiers à consulter pour connaître les dernières modifications du système.
INSTALL*.BAT	Programmes d'installation du logiciel.
***.ARC**	Fichiers compactés contenant des exemples en Turbo Prolog.
UNPACK.EXE	Utilitaire de décompactage des fichiers *.ARC.

Disquette 2/4

PROLOG.EXE	Le compilateur.
BGI*.*	Fichiers graphiques.

Disquette 3/4

PROLOG.ERR	Les messages d'erreurs.
PROLOG.HLP	Fichiers en ASCII d'aide à l'utilisateur.
PROLOG.OVL	Fichiers de recouvrement du compilateur.
***.BGI et*.CHR**	Fichiers graphiques.

Disquette 4/4

PROLOG.LIB et INIT.OBJ

Fichiers utilisés par Turbo Prolog pour générer des programmes exécutables sous DOS (extension .EXE).

2.1.3. Extension des fichiers crées sous Turbo Prolog

.PRO donnée par défaut aux programmes créés sous Turbo Prolog. L'utilisateur peut définir une extension de son choix ou pas d'extension. (le nom du fichier doit alors se terminer par un point)
.BAK affectée par Turbo Prolog à la version primitive du programme lors d'une nouvelle sauvegarde.
.EXE code objet (programme compilé) exécutable sous MSDOS.
.SYS fichier de configuration du système créé à partir du menu Setup.

2.1.4. Installation du logiciel

Turbo Prolog n'est pas un logiciel protégé, il est donc vivement conseillé d'en faire une copie une copie de sauvegarde.
Que vous ayez un disque dur ou deux lecteurs de disquettes, il faudra installer le logiciel pour pouvoir l'utiliser.

Avec deux lecteurs de disquettes :

- Préparez quatre disquettes formatées,
- Placer la disquette 1/4 dans le lecteur A, une disquette en B,
- Taper depuis le prompt du Dos :

```
A:> install a: b: ↵
```

- suivez les directives données à l'écran.

Avec un disque dur :

- Placer la disquette 1/4 dans le lecteur A,
- Taper depuis le prompt du Dos :

```
C:> a:install a: c:\prolog ↵
```

"prolog" est le nom du répertoire dans lequel vous désirez installer Prolog (vous pouvez en définir un autre),
- Suivez alors les directives affichées à l'écran.

Mise à jour des fichiers CONFIG.SYS et AUTOEXEC.BAT
Pour que Turbo Prolog puisse fonctionner le fichier CONFIG.SYS doit contenir les paramètres :

```
FILES = 20
BUFFERS = 40
```

Si vous voulez pouvoir appeler Turbo Prolog depuis n'importe quel répertoire du disque, ajouter dans le fichier AUTOEXEC.BAT le chemin correspondant :
Exemple

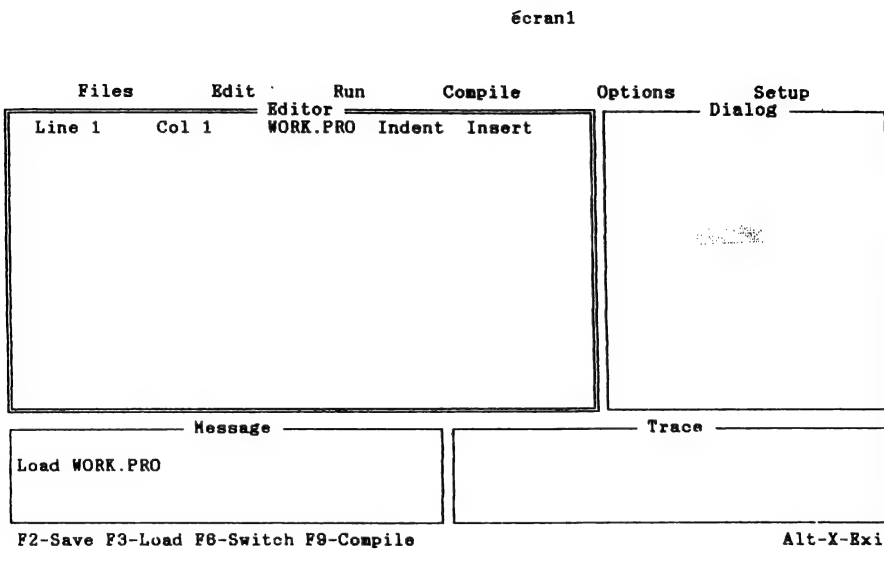
```
PATH c:\PROLOG
```

22.2. MISE EN OEUVRE DE TURBO PROLOG

Taper à partir du système d'exploitation : PROLOG ↵

2.2.1. Menu principal

L'écran suivant est généré :



L'écran est divisé en quatre fenêtres :

Editor
C'est dans cette fenêtre que s'afficheront les programmes.
Dialog
C'est à travers cette fenêtre que se feront les échanges d'informations entre l'utilisateur et le programme.
Message
Prolog transmettra à l'utilisateur des informations sur le déroulement du programme à travers cette fenêtre.
Trace
C'est dans cette fenêtre que l'utilisateur pourra suivre le tracé pas à pas de l'exécution d'un programme.

La ligne du haut de l'écran correspond au menu principal :

- **Edit** active la fenêtre d'édition.
- **Run** exécute le programme en cours d'édition (si celui-ci a été modifié depuis la dernière compilation, il est préalablement compilé).
- Les autres commandes **Files**, **Compile**, **Options** et **Setup** déroulent un sous menu.

On active une commande du menu principal depuis le menu principal, soit :

- en déplaçant le pointeur lumineux sur l'option choisie, puis en enfonçant ↵
- en tapant la première lettre de l'option.

Pour activer une sous-commande d'un sous-menu, activer le sous menu et procéder comme pour une commande.
- **Esc** permet de quitter le menu actif pour retourner au menu de niveau supérieur.
- **F10** ramène directement au menu principal.

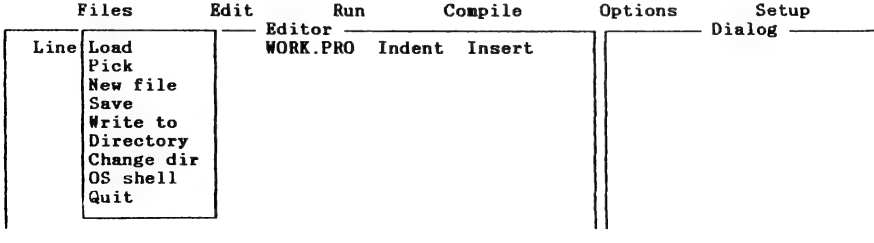
La dernière ligne de l'écran donne la signification des touches fonctions : celles-ci diffèrent suivant la commande active.

Il n'a pas paru utile de redonner systématiquement, dans ce manuel, les actions de ces touches, la dernière ligne de l'écran les affiche à tout moment et est suffisamment explicite. Lorsque une fenêtre est active, les touches de déplacement du curseur modifient ses dimensions et F5 (zoom) l'agrandit à tout l'écran.

A tout moment, il est possible d'activer directement un sous-menu sans repasser par le menu principal en enfonçant simultanément Alt et la première lettre de l'option choisie.

Il existe trois sortes de commandes dans l'environnement intégré :
- les commandes pour accomplir une tâche.
- les commandes interrupteurs : On/Off.
- les commandes pour transmettre des informations au compilateur : répertoires, directives etc..

2.2.2. Menu Files



Load
Charge un fichier en mémoire. Par défaut Load donne les fichiers *.PRO. Il est possible d'utiliser un masque du DOS, puis de choisir un fichier dans la liste, ou de spécifier directement un nom de fichier. Pour charger un fichier d'un autre répertoire que le répertoire actif, déplacer le pointeur lumineux sur le répertoire voulu et enfoncer ↵

Pick
Affiche la liste des huit derniers fichiers chargés dans l'éditeur. Pour charger l'un de ces fichiers, placer le pointeur lumineux sur le fichier voulu et enfoncer ↵

New File
Efface l'éditeur et crée un nouveau fichier WORK.PRO.
Si le fichier n'est pas sauvé, Prolog demande confirmation avant de l'effacer.

Save
sauve le fichier actif dans l'éditeur. Même action que F2.

Write to
Sauve le fichier de l'éditeur sous un nouveau nom.

Directory
affiche le contenu (*.* par défaut) du répertoire courant.Il est possible d'utiliser un masque du DOS.

Change dir
Permet de changer d'unité active et/ou de répertoire courant.

OS shell
permet d'aller sous l'interpréteur de commandes (COMMAND.COM) du DOS, puis de revenir sous Turbo Prolog en tapant "exit".

Quit
Retour sous Dos. Analogue à Alt X.

2.2.3. Commande Edit

La première ligne de l'éditeur donne des informations sur l'édition :

Line n	numéro de ligne du curseur
Col n	numéro de colonne du curseur
Indent	mode indentation actif
Insert	mode insertion actif

Turbo Prolog affecte par défaut le nom WORK.PRO au fichier en cours d'édition.

Déplacement du curseur :

 déplacement d'un caractère dans la direction indiquée.

`PgUp`	écran précédent.
`PgDn`	écran suivant.
`Home`	début de ligne.
`End`	fin de ligne.
`Ctrl Q R`	début de fichier.
`Ctrl Q C`	fin de fichier.
`Ctrl Q B`	début de bloc.
`Ctrl Q K`	fin de bloc.
`Ctrl Q P`	dernière position du curseur.
`Ctrl O T`	pose de tabulations.

Effacement :

`Ctrl G` ou `Del`	efface le caractère sous le curseur.
`<-`	efface le caractère à gauche du curseur.
`Ctrl T`	efface le mot à droite du curseur.
`Ctrl Y`	efface la ligne du curseur.

Commandes de modifications :

`Ctrl N`	insère une ligne à la position du curseur.
`Ins`	bascule du mode insertion en recouvrement et vice-versa.
`Ctrl Q I`	supprime ou active l'indentation.

Utilisation des blocs :

`Ctrl K B`	marque un début de bloc.
`Ctrl K K`	marque une fin de bloc (le bloc marqué apparait en sous brillance).
`Ctrl K H`	supprime les repères de bloc.
`Ctrl K V`	déplace le bloc à la position du curseur.
`Ctrl K C`	duplique le bloc à la position du curseur.
`Ctrl K Y`	détruit le bloc.
`Ctrl K R`	lecture d'un bloc depuis le disque (donner après avoir introduit la commande le nom du fichier à lire).
`Ctrl K W`	écriture d'un bloc sur le disque (donner après avoirintroduit la commande le nom du fichier à créer ou recréer).
`Ctrl K P`	impression du bloc marqué.

Recherche et remplacement :

`Ctrl Q F` Recherche
`Ctrl Q A` Recherche et remplace

Aide sur le langage :

`Ctrl F1`

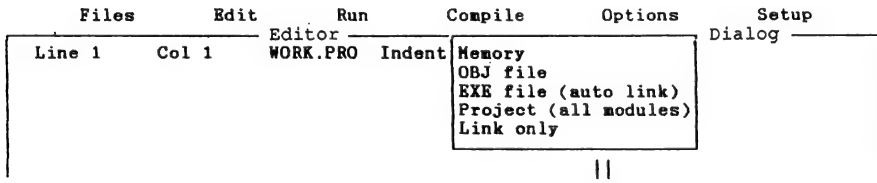
2.2.4. Commande Run

La commande Run compile (s'il y a lieu) puis exécute le programme en mémoire.
S'il existe un but interne, il est exécuté, sinon vous pouvez entrer tous les buts externes que vous désirez dans la fenêtre du dialogue.

Lorsque la commande RUN est active :

F8	réaffiche dans la fenêtre de dialogue le but précédent.
F9	appelle l'éditeur.
F10	reconfigure la fenêtre sélectionnée.
Ctrl S	suspend momentanément l'exécution du programme. Enfoncer n'importe quelle touche pour reprendre l'exécution.
Ctrl Break (ou Ctrl C)	interrompt l'exécution du programme.
Alt P	imprime l'affichage en écho sur l'imprimante. Enfoncer à nouveau Ctrl P pour arrêter la sortie sur l'imprimante.

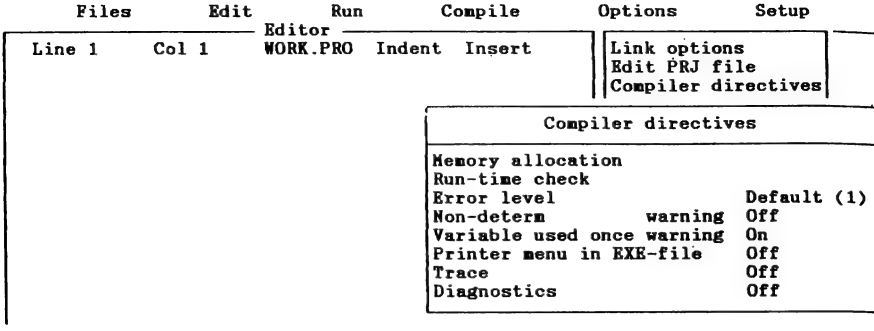
2.2.5. Commande Compile



Cette commande compile le programme en cours d'édition. Le menu Options permet d'indiquer à Turbo Prolog ce que l'on désire en matière de compilation. Par défaut, la compilation génère un programme en mémoire vive. Pour générer un programme exécutable sous DOS (.EXE), sélectionner l'option EXE.
Les autres options ne sont utiles que pour les programmeurs désirant intégrer des routines écrites dans un autre langage dans leur programme.

2.2.6. Menu Options

Seule l'option **Compiler directives** de ce menu est couramment utilisée :



Cette commande permet de spécifier depuis l'environnement de Turbo Prolog les directives de compilation :

Memory allocation
permet de spécifier les dimensions (en paragraphes de 16 octets) du code, de la pile et du tas. Par défaut la taille de la pile est de 600 paragraphes de 16 octets. Si votre programme le nécessite (Stack overflow) vous pouvez spécifier une dimension supérieure, pouvant aller jusqu'à 4000 paragraphes.

Run-time check
permet de spécifier au compilateur des vérifications sur l'état de la touche Break, l'occupation de la pile et les débordements d'entiers. Si ces options sont On, l'exécution des programmes sera ralentie, elles sont à activer au cours de la mise au point des programmes.

Error Level
permet de choisir le niveau d'informations que l'on désire obtenir en cas d'erreurs à l'exécution. Par défaut le niveau est celui intermédiaire.

Non-Determ Warning
signale la présence de clauses non déterministes, c'est à dire, celles qui donnent plusieurs solutions.

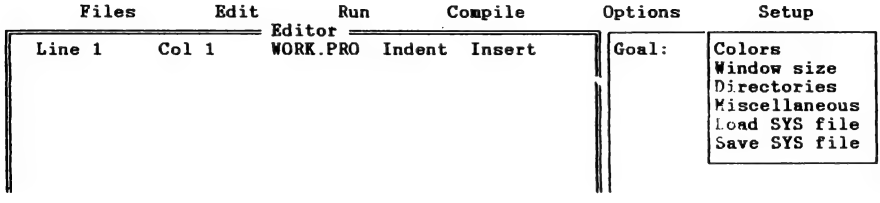
Variable Used Once Warning
signale les variables utilisées une seule fois dans une clause (active par défaut).

Printer Menu in EXE-file
Lorsque cette directive est active, il est possible d'utiliser les touches Alt P pour sortir l'exécution d'un programme compilé sur l'imprimante.

Trace
permet de suivre l'exécution du programme pas à pas, sans avoir à insérer la directive Trace dans le programme, c'est fort utile !

Diagnostics
Lorsque cette directive est active, à chaque exécution du programme, Turbo affichera un diagnostic de celui-ci.

2.2.7. Menu Setup

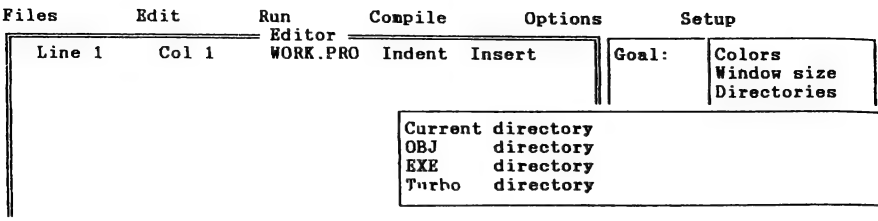


Ce menu permet de modifier les caractéristiques que Turbo Prolog affecte par défaut au matériel.

Colors
Cette option permet de fixer les couleurs de premier plan et de fond de chaque fenêtre. La sélection de cette commande génère un sous menu correspondant aux fenêtres de Turbo Prolog.
Sélectionner une fenêtre et modifier les couleurs en utilisant les touches :
<↑↓> pour l'avant plan (les caractères)
pour le fond.

Windows size
correspond à une autre méthode pour modifier la dimension et également l'emplacement des fenêtres. La sélection de cette commande génère un sous menu.
Sélectionner une fenêtre et modifier ses caractéristiques en utilisant les touches :
<↑↓> rétrécir, agrandir la fenêtre en largeur.
rétrécir, agrandir la fenêtre en hauteur.Num
Lock et la touche concernée déplacer la fenêtre.

Directories



Cette option permet de spécifier plusieurs répertoires dans lesquels Turbo Prolog sauvera automatiquement les fichiers créés suivant leur catégorie. Pour les fichiers supportant plusieurs chemins, ceux-ci sont énumérés, séparés par des points virgules ";".

Current Directory est le répertoire depuis lequel Turbo Prolog chargera et sauvera, par défaut, les programmes .PRO.
OBJ Directory pour les fichiers .OBJ
EXE Directory pour les fichiers .EXE
Turbo Directory contient le système Turbo Prolog (PROLOG.EXE, PROLOG.ERR, PROLOG.HLP, PROLOG.OVL PROLOG.LIB et INIT.OBJ)

Miscellaneous

ou arrangements divers, fait apparaître un menu avec cinq options :

IBM CGA Adapter
permet d'adapter Turbo Prolog à l'écran CGA.

Auto Load Message
détermine si le fichier d'erreurs doit être chargé en mémoire (On) ou si chaque message est extrait de TURBO.ERR quand il est nécessaire (Off).

Screen mode
permet de changer le mode d'affichage par défaut (25 lignes, 8 colonnes).

Keyboard configuration
permet de redéfinir les touches fonction et les commandes de l'éditeur.

Help lines
Cette option permet d'afficher sur la dernière ligne de l'écran la signification des touches fonctions pour chaque menu sans avoir à activer le menu correspondant.

Load SYS file et Save SYS file

Lorsque vous avez configuré votre système à votre convenance (c'est à dire, fixés les couleurs, les tailles et position des fenêtres et définis les répertoires) vous pourrez sauver la configuration dans un fichier .SYS en utilisant l'option : Save Configuration. Par défaut, ce fichier est TURBO.SYS.
A chaque initialisation, Turbo Prolog charge automatiquement le fichier TURBO.SYS, lequel est obligatoirement dans le répertoire TURBO.
Si vous nommez différemment votre fichier .SYS, vous devrez le charger vous-même par la commande : Load Configuration.

2.3. PRESENTATION GENERALE DU LANGAGE

2.3.1. Structure d'un programme

Un programme en Turbo Prolog est articulé en quatre sections, identifiée chacune par un mot clé et se présentant dans la séquence suivante :

```
domains /* déclarations des types */
predicates /* déclarations des prédicats */
goal /* but ou suite de buts */
clauses /* faits et règles */
```

Les symboles `/* */` encadrent les commentaires. Les deux premières parties (`domains` et `predicates`) correspondent aux déclarations du Pascal. La section `clauses` contient la base de connaissances. Toutes les sections sont optionnelles. Ainsi un programme pourrait n'être qu'une section `Goal`.

Par exemple :

```
goal
   write("Bonjour").
```

Notez le point "." obligatoire à la fin de la section `goal`. Le plus souvent, un programme sera constitué des sections `predicates` et `clauses`.

2.3.2. Catégories d'objets

Turbo Prolog manipule deux sortes d'objets : les constantes et les variables. Ces objets peuvent eux-même être simples ou composés. Les objets composés (ou complexes) seront étudiés au chapitre correspondant.

Les **constantes** représentent des connaissances absolues. Une constante commence obligatoirement par une lettre minuscule.

Exemple : "Verlaine est un poète", s'écrira en Turbo Prolog :

```
poète(verlaine)
```
verlaine est une constante.

Les **variables** désignent des objets dont nous connaissons l'existence mais dont la valeur n'est pas fixée à l'instant présent. Une variable commence obligatoirement par une lettre majuscule.

Exemple : `poète(X)` signifie que X est un poète.

Tant qu'une variable attend de recevoir une valeur, elle est dite *libre* : elle est inconnue. Lorsqu'elle reçoit une valeur, elle est dite *liée* à cette valeur : elle devient une constante.
Au cours de l'exécution d'un programme une variable est tantôt libre, tantôt liée.

Variables anonymes

Turbo Prolog permet de signaler la présence d'une variable dont la valeur nous importe peu, par l'utilisation des variables anonymes. Une variable anonyme est représentée par le caractère blanc souligné "_".

Exemple : `poète(_)` désigne l'existence d'un poète dont le nom nous indiffère.

2.3.3. Types des objets simples

Tout objet manipulé par Turbo Prolog doit obligatoirement être identifié par son type. Les déclarations de type sont regroupées dans la section `domains`.

1) Syntaxe des déclarations de type simple :

```
domains
 objet11, objet12, ... = type1
 objet21, objet22, ... = type2
 ...
```

2) Types simples prédéfinis

char
définit un des 256 caractères ASCII figurant entre apostrophes.

Exemple : 'a' est de type char.

integer
tout entier compris entre -32768 et 32767.

real
tout nombre réel compris entre 1E-308 et 1E+308.

sting
toute suite de caractères entre guillemets.

Exemple : "Bonjour" est de type `string`.

symbol
suite de caractères, sans espace, commençant par une minuscule et comportant d'éventuels caractères soulignés ou suite de caractères entre guillemets.

Exemple :
```
victor_hugo
"Victor Hugo"
```
Attention à la syntaxe :
"a" est une chaîne ou un symbol, alors que :
'a' est un caractère
`string` et `symbol` sont interchangeables, la forme symbol est plus rapide à l'exécution mais prend plus de place.

file
déclaration de nom symbolique de fichier pour la durée du programme (le fichier DOS porte un autre nom).

Exemple :
```
file = fichier_essai
```

2.3.4. Prédicats et clauses

L'énoncé d'une proposition (fait ou règle) se fait en deux étapes :
- déclaration du prédicat.
- énoncé d'une ou plusieurs clauses illustrant le prédicat.

1) Prédicats
La déclaration d'un prédicat est semblable à la déclaration d'une fonction ou d'une procédure en Pascal.
Un prédicat énonce une relation entre ses arguments. Le type de chaque argument doit avoir été préalablement déclaré.
Un prédicat doit toujours être illustré par au moins une clause.

2) Clauses
Les clauses doivent toujours se référer à un prédicat préalablement déclaré. On dit qu'une clause valorise un prédicat.
Toutes les clauses relatives à un même prédicat doivent être regroupées.

Exemple :

```
domains
    qqn,qqc = symbol

predicates
    possède(qqn,qqc)

clauses
    possède(jean,voiture).
    possède(pierre,maison).
    possède(jean,bateau).
```

Remarquer le point "." obligatoire à la fin de chaque clause.
Le prédicat "possède" énonce une relation entre "qqn" et "qqc", lesquels sont précisés dans les clauses. Turbo Prolog sait que "jean" et "pierre" sont "qqn" et "voiture", "maison" et "bateau" sont "qqc" à cause de leur emplacement dans les clauses.

Lorsque les objets sont de type simple la section domains peut être omise et la déclaration de type a lieu dans le prédicat.

Exemple :

```
predicates
   possède(symbol,symbol)
```

Si vous désirez commencer les noms propres par une majuscule, utiliser les guillemets.

Exemple :

```
possède("Jean",voiture).
```

2.3.5. Formuler un but

Turbo Prolog fonctionnant en chaînage arrière, il faut lui fixer un but soit externe (par la fenêtre de dialogue au moment de l'exécution) soit interne (fixé dans la section goal).
La formulation des interrogations (requêtes) par but externe ou interne est identique.
Turbo Prolog donne trois types de réponses à une requête : Yes, No ou les solutions.
- Yes correspond à une proposition vérifiée.
- No à une proposition fausse.
- Les solutions seront affichées lorsque la requête contient des variables.

2.3.6. Exemple : une base de données relationnelle

```
predicates

    aime(symbol,symbol)
    travaille(symbol,symbol,integer)
```

```
clauses
    aime(pierre,pêcher).
    aime(jean,nager).
    aime(jean,chasser).
    aime(charlotte,danser).
    aime(constance,danser).
    travaille(jean,mathématiques,6).
    travaille(jean,français,4).
    travaille(pierre,informatique,8).
    travaille(charlotte,droit,5).
```

Nous avons constitué une base de données sur les loisirs et les temps d'étude d'un petit groupe d'individus. Entrer ce programme et faites le tourner en sélectionnant la commande `Run`. Le système affiche dans la fenêtre de dialogue :

`Goal : _` (Il attend une requête)

En français nous demanderons par exemple :
Jean aime-t-il nager ?
En Prolog, nous écrirons :

```
aime(jean,nager)
```

La machine répond : `Yes`, puisque la proposition est vraie.
Si nous demandons :

```
aime(jean,danser)
```

La machine répond : `No`
En effet dans la logique de Turbo Prolog toute assertion est nécessairement vraie ou fausse et tout ce qui n'est pas expressément vrai est réputé faux. Aucune clause ne spécifiant ici que `"jean aime danser"`, Turbo Prolog répond que cette proposition est fausse.

Utilisons maintenant notre base pour poser une question du genre :
Qu'aime faire Pierre de ses loisirs ?
En Prolog, nous écrirons :

```
aime(jean,Quoi)
```

et nous obtenons :

```
Quoi = nager
Quoi = pêcher
2 solutions
```

Le "Q" majuscule est impératif pour indiquer qu'il s'agit d'une variable. Un "q" minuscule donnerait la réponse No, car la question serait interprétée par la machine sous la forme :

Jean aime-t-il le loisir intitulé quoi ?
La machine ayant reconnu une variable, la base de données va être explorée à la recherche des clauses illustrant la requête.
Toutes les clauses "aime" ayant pour premier paramètre "jean" répondent à la question. La variable "Quoi" se voit affectée les valeurs satisfaisant la requête et les solutions correspondantes sont affichées.

Utilisons la base pour poser une question du genre : Pierre aime-t-il quelque chose ?
En Turbo Prolog nous écrirons :

```
aime(pierre,_)
```

La machine répond : Yes. En effet, il est vrai que Pierre aime quelque chose et nous ne désirons pas connaître cette chose. Nous désirons maintenant poser deux questions simultanément. Par exemple :

Qu'aime faire Pierre de ses loisirs et quelles matières travaille-t-il et pendant combien de temps ?

En Turbo Prolog, nous écrirons :

```
aime(pierre,Quoi),
travaille(pierre,Quel,Combien)
```

Prolog affiche :

```
Quoi = pêcher, Quel = informatique,
Combien = 8
1 Solution
```

Remarquer la virgule "," qui sépare les deux sous buts. Elle correspond à l'opérateur logique AND que nous verrons dans le prochain chapitre. Utiliser la base créée pour formuler de nouvelles requêtes, par exemple :

```
aime(Qui,Quoi)
aime(_,_)
```

Ce qui correspond, en français aux questions :

Qui aime quoi ?
Quelqu'un aime-t-il quelque chose ?

2.4. REGLES

2.4.1. Définition

Une règle exprime une relation conditionnelle entre des faits. Elle se compose de deux parties séparées par la condition SI :

Tête de règle SI Queue de règle

Ce qui signifie : La tête de règle est vraie SI la queue de règle est vraie. La tête de règle (ou conclusion) est un but (un fait) unique illustrant un prédicat. La queue de règle est composée d'un ou plusieurs faits (ou sous buts) reliés par des ET et des OU. Une règle peut être perçue comme une procédure dont le membre de gauche représente l'en-tête et le coté droit le corps, c'est à dire la séquence des instructions à exécuter dans un ordre déterminé.

2.4.2. Syntaxe

Soit un prédicat r et une suite de buts représentés par : $fait_{ij}$. Les règles r_i du prédicat r seront de la forme :
r_i IF $fait_{i1}$ AND $fait_{i2}$ AND ... $fait_{in}$. Chaque règle doit obligatoirement se terminer par un point.

Notation
Turbo Prolog autorise les notations simplifiées suivantes :

IF	:-
AND	,
OR	;

Ces notations seront celles utilisées dans ce manuel.

2.4.3. Alternative OR

L'alternative OR est relativement peu utilisée en Prolog. On préférera en général illustrer un prédicat par plusieurs règles. Si nous voulons par exemple, exprimer qu'un prédicat r est vrai si l'un des sous buts a OU b est vrai, nous pourrons écrire :
r :- a ; b.
Mais on préférera généralement créer deux règles r_1 et r_2 de la forme :
r_1 :- a.
r_2 :- b.

2.4.4. Opérateurs de comparaison

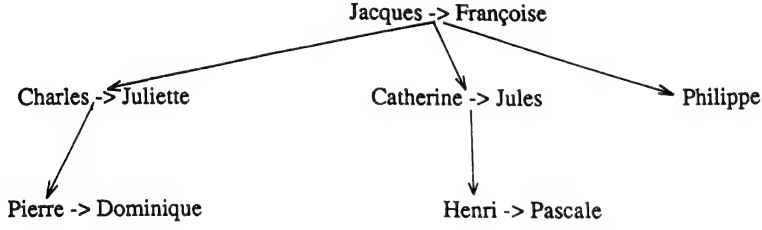
<	inférieur à
>	inférieur ou égal à
=	égal à
>=	supèrieur à
>=	supérieur ou égal à
<> ou ><	différent de

Elles permettent de comparer entre eux des nombres, des caractères, des chaînes de caractères et des symboles. (le classement des caractères est bien entendu le classement ASCII).

2.4.5. Exemple : un premier système expert

Le but de notre système expert est la réalisation d'un arbre généalogique. La base des faits décrit la famille, la base des régles établit les relations entre les individus.

La famille :




```
predicates

    homme(symbol)
    femme(symbol)
    père(symbol,symbol)
    conjoint(symbol,symbol)
    mère(symbol,symbol)
    enfant(symbol,symbol)
    parent(symbol,symbol)
    frère(symbol,symbol)
    soeur(symbol,symbol)
    oncle(symbol,symbol)
    mari(symbol,symbol)
    épouse(symbol,symbol)
```

```
clauses

   /* Base de faits */

   homme("Jacques").
   homme("Charles").
   homme("Jules").
   homme("Pierre").
   homme("Henri").
   homme("Philippe").

   femme("Françoise").
   femme("Juliette").
   femme("Catherine").
   femme("Pascale").
   femme("Dominique").

   père("Charles","Jacques").
   père("Catherine","Jacques").
   père("Pierre","Charles").
   père("Henri","Jules").
   père("Philippe","Jacques").

   conjoint("Jacques","Françoise").
   conjoint("Charles","Juliette").
   conjoint("Pierre","Dominique").
   conjoint("Henri","Pascale").
   conjoint("Catherine","Jules").

   /* Base de règles */

/*Y est l'épouse de X si Y est une femme et si Y est le
conjoint de X ou l'inverse*/
   épouse(X,Y):-
femme(Y),conjoint(X,Y).
épouse(X,Y):-
femme(Y),conjoint(Y,X).

   /*Y est le mari de X si Y est l'épouse de X*/
   mari(X,Y):-
épouse(Y,X).

/*Y est la mère de X si Z est le père de X et Y et Z sont
mariés*/
   mère(X,Y):-
père(X,Z),mari(Y,Z).
```

```
/*Y est l'un des parents de X si Y est son père ou sa mère*/
   parent(X,Y):-
père(X,Y).
   parent(X,Y):-
mère(X,Y).

   /*Y est enfant de X si X est le père ou la mère de X*/
   enfant(X,Y):-
père(Y,X).
   enfant(X,Y):-mère(Y,X).

/*le frère de X est Y si: Y est un homme et le père de X est
le père de Y et X<>Y */
frère(X,Y) :-
homme(Y),père(X,P),père(Y,P),X <> Y.

   soeur(X,Y):-
femme(Y),père(X,P),père(Y,P),X <> Y.

/*Y est l'oncle de X si un parent de X a pour frère Y ou si un
parent de X a une soeur dont le mari est Y*/
   oncle(X,Y):-
parent(X,P),frère(P,Y).
   oncle(X,Y):-
parent(X,P),soeur(P,Q),mari(Q,Y).
```

Exemples de requêtes

- Quel est le père de Charles ? se formulera :

```
père("Charles",X)
```

- Quelles sont les soeurs d'Henri ? se formulera :

```
soeur("Henri",X)
```

- Catherine est-elle la soeur de Jacques ? se formulera :

```
soeur("Jacques","Catherine")
```

Formuler vous même d'autres requêtes et ajouter à la base des relations nouvelles : grand-père, grand-mère, petits enfants, descendant, cousin, etc.

2.5. OUTILS D'AIDE A LA PROGRAMMATION

Comme pour tous les langages de programmation, les erreurs commises dans un programme peuvent être syntaxiques ou logiques.

2.5.1. Turbo Prolog détecte les erreurs de syntaxe

Lorsque Turbo Prolog a détecté une erreur de syntaxe un message apparaît au bas de la fenêtre d'édition (si le fichier PROLOG.ERR est présent dans le même répertoire que PROLOG.EXE), le curseur se place sous la faute et clignote, le mode d'édition est automatiquement activé. Les messages d'erreurs sont si explicites que des commentaires supplémentaires dans ce manuel ont paru inutiles. Aucun langage de programmation ne possède une syntaxe aussi simple et concise que Turbo Prolog.
Rappelons brièvement quelques règles syntaxiques :

- La section `goal` doit se terminer par un point ".".
- Chaque clause doit se terminer également par un point ".".
- Les clauses relatives à un même prédicat doivent être regroupées.
- Une variable doit obligatoirement commencée par une lettre majuscule, une constante par une minuscule. Les autres éléments constitutifs d'un programme peuvent être indifféremment en minuscules ou en majuscules.
- Le caractère espace est traditionnellement représenté par le caractère souligné "_".

2.5.2. Turbo Prolog détecte certaines erreurs de logique

En plus de la détection des erreurs de syntaxe, Turbo Prolog signale au programmeur les erreurs de logique qu'il est capable de détecter, comme : les variables non liées dans une clause ou les variables libres dans une expression...
Une fois le programme syntaxiquement correct, il appartient au programmeur d'en tester la logique.

Deux prédicats prédéfinis peuvent être utiles :

`bound(Variable)`
qui réussit si Variable est liée.

`free(Variable)`
qui réussit si Variable est libre.

2.5.3. Utiliser les buts externes

L'utilisation d'un but externe est un outil d'aide à la programmation. Vous pourrez, en choisissant des buts externes judicieux, tester les divers prédicats que vous avez créés, même après la compilation du programme.

2.5.4. Suivre un programme pas à pas

La directive de compilation `trace` permet de suivre pas à pas le déroulement d'un programme dans les fenêtres Trace, Editor et Dialog.
Lancer l'exécution du programme à tracer. A chaque fois que vous enfoncerez F10 vous déclencherez le fonctionnement en pas à pas du programme. Suivez alors ce qui est affiché dans la fenêtre Trace et également les déplacements du curseur dans l'éditeur. Lorsque `trace` est spécifiée, la fenêtre de dialogue affichera successivement :

`CALL:.`.
suivi du nom du prédicat appelé et de la liste des paramètres entre parenthèses, séparés par des virgules. Les paramètres libres sont représentés par le caractère souligné "_" (variable anonyme), les paramètres liés sont remplacés par leurs valeurs.

`RETURN:.`.
apparaît lorsqu'un prédicat a réussi.
Le nom du prédicat qui a réussi suit RETURN avec les valeurs des paramètres.
S'il existe d'autres clauses du prédicat à vérifier, Turbo Prolog fait suivre RETURN d'un astérisque "*" qui indique donc que la remontée va être amorcée.

`FAIL:.`.
apparaît suivi du nom du prédicat qui a échoué avec la liste des paramètres du dernier appel du prédicat.

`REDO:.`.
indique que la remontée est en cours. L'écran affiche le nom du prédicat que le système essaie à nouveau de satisfaire, ainsi que la liste de ses paramètres.

2.5.5. Recherche des solutions par Turbo Prolog

Comme nous l'avons déjà signalé Turbo Prolog est un moteur d'inférence d'ordre un fonctionnant en chaînage arrière.
Le mode `trace` permet de suivre pas à pas la recherche des solutions et par conséquent d'expliciter parfaitement les deux mécanisme de base du moteur d'inférence : l'unification et la remontée.

Unification

Consiste à déterminer les valeurs que peuvent prendre les variables au cours de la résolution, cela correspond au passage de paramètres dans d'autres langages.
L'unification entre deux objets Prolog est possible :
- soit, si les deux objets sont identiques.
- soit par substitution des éléments variables de l'un des objets par les éléments correspondants fixés ou variables de l'autre objet. Ce que l'on peut traduire par les règles :
- une variable libre est unifiable à n'importe quel terme.
- une constante est unifiable à elle-même ou à une variable libre.

- deux objets complexes sont unifiables s'ils ont le même foncteur et le même nombre d'arguments. Les listes sont des objets complexes particuliers (voir "Objets complexes").

Exemple :

- `nombre(100)` et `nombre(200)` ne sont pas unifiables.
- `aime(X,Y)` et `aime(jean,chanter)` sont unifiables.

Pour comprendre les exemples suivants, vous devez avoir préalablement étudié les chapitres sur les objets complexes et les listes.

- `livre(auteur(X,Y),T)` et `livre(auteur(hergé,_),le_lotus_bleu)` sont unifiables.
- `Liste et [a,b,c]` sont unifiables.
- `[X] et [a,b,c]` ne sont pas unifiables, la première liste ne contient qu'un seul élément.
- `[X:Y]` et `[a,b,c]` sont unifiables par X = a et Y = [b,c]
- `[X,Y]` et `[a,b,c]` ne sont pas unifiables.

Remontée

Le programme suivant exprime qu'un individu habite une capitale s'il habite une ville et que cette ville est une capitale.

```
trace
predicates
 habite(symbol,symbol)
 capitale(symbol)
 habite_capitale(symbol)

clauses
 habite(jean,paris).
 habite(pierre,marseille).
 habite(john,londres).
 habite(peter,washington).

 capitale(paris).
 capitale(londres).
 capitale(washington).

 habite_capitale(pedro).
 habite_capitale(juan).
 habite_capitale(X):-habite(X,Y),capitale(Y).

Trace du programme obtenue en entrant le but :
 habite_capital(Qui)
CALL:  habite_capitale(_)
RETURN:*habite_capitale("pedro")    Ire  solution
REDO:  habite_capitale(_)
RETURN:*habite_capitale("juan")     2ème solution
REDO:  habite_capitale(_)
```

```
CALL:  habite(_,_)
RETURN:*habite("jean","paris")      1er sous but satisfait
CALL:  capitale("paris")
RETURN: capitale("paris")           2d sous but satisfait
RETURN habite_capitale("jean")      3ème solution
REDO:  habite(_,_)
RETURN:*habite("pierre","marseille")
CALL:  capitale("marseille")
REDO:  capitale("marseille")
REDO:  capitale("marseille")
FAIL:  capitale("marseille") Echec
REDO:  habite(_,_)
...
```

Le scénario se poursuit de manière similaire pour les deux autres solutions.

Pour satisfaire un but, Turbo Prolog explore la base de faits de haut en bas à la recherche d'une solution.

- Turbo Prolog commence par chercher le prédicat identique à la requête, puis teste les clauses de ce prédicat dans leur ordre d'apparition dans le programme : les deux premières solutions trouvées dans l'exemple sont `pedro` et `juan`.
- Quand un but coïncide avec la tête d'une règle, Turbo Prolog cherche à satisfaire les prémisses de la règle : lorsque le programme fait coïncider `habite_capitale(_)` avec la troisième clause de ce prédicat il cherche a satisfaire les sous buts `habite et capitale`.
- Les sous buts d'un règle sont satisfaits de gauche à droite.

C'est uniquement en cas de succès du premier sous but que Turbo Prolog passera au suivant : le programme satisfait `habite`, ce qui lie la variable Y et ensuite cherche si Y est une capitale.

- Un but est atteint quand un fait est découvert pour toutes ses extrémités : les solutions correspondent aux deux faits du prédicat `habite_capitale` et aux faits satisfaisants simultanément les sous buts `habite(X,Y)` et `capitale(Y)`.

Jeu combiné de l'unification et de la remontée

Lorsque Turbo Prolog cherche à satisfaire un but (ou un sous but) utilisant des variables, il cherche à unifier le premier argument du prédicat concerné avec une valeur trouvée dans la base de faits et place un marqueur dans la base indiquant "c'est ici que j'ai lié une valeur à une variable". Puis descendant la base de faits, il cherche à satisfaire le second argument, s'il trouve correspondance, il y a succès : la recherche se poursuit pour l'argument suivant. S'il y a échec, la remontée s'effectue jusqu'à la position du dernier marqueur. La variable est libérée et la recherche reprend au fait suivant avec une nouvelle unification de la variable.

2.5.6. Comment limiter la quantité d'informations générées par trace ?

L'utilisation de la directive `trace` peut générer trop d'informations lors de la mise au point de programmes volumineux. Plusieurs solutions s'offrent au programmeur :

- Utiliser la directive `shorttrace` au lieu de `trace` qui limite les messages dans la fenêtre Trace.
- Spécifier après l'une des directives `trace` ou `shorttrace` le nom des prédicats dont on veut suivre la trace :

```
trace p1, p2, p3,...
```

- Utiliser le prédicat `trace(on/off)`.
Ce prédicat réussit toujours. `trace(on)` active le mode trace (l'une des directives `trace` ou `shortttrace` doit être activée). `trace(off)` désactivera le mode trace. On peut ainsi délimiter une portion de programme à tracer.

2.6. RENDRE UN PROGRAMME CONVIVIAL

2.6.1. But externe - But interne

Considérons le programme :

```
predicates
 possède(symbol,symbol)

clauses
 possède("Jean",voiture).
 possède("Jean",livre).
 possède("Pierre",logement).
 possède("Nicole",livre).
```

Afin de rendre ce programme plus convivial, nous serons probablement tentés d'ajouter une section :

```
goal
 possède(X,Y).
```

après la section `predicates`. Or nous constatons après avoir lancer l'exécution du programme par "`run`" que rien ne s'affiche. Que s'est-il passé ? Un programme complet en Turbo Prolog aura toujours une section `goal`. L'utilisation d'un but externe contenant des variables est en fait une aide à la programmation et provoque l'affichage de toutes les solutions alors que par but interne, c'est le programmeur qui doit gérer la recherche et l'affichage des solutions lui-même. C'est à dire forcer Turbo Prolog à rechercher toutes les solutions et à les afficher.

2.6.2. Entrées/sorties

Comme tous les langages de programmation, Turbo Prolog possède des instructions (des prédicats prédéfinis appelés plus simplement des prédéfinis) d'entrée/sortie qui permettent de fournir des données au programme et de récupérer des résultats.

Ecriture

```
write(arg1,arg2,arg3,...)
```

affiche les arguments spécifiés à l'écran. Les arguments (en nombre quelconque) sont des constantes ou des variables liées. Le caractère de contrôle "\n" provoque un renvoi à la ligne.

Exemple :

```
write("Ca marche\n")
```

```
nl
```
provoque un saut de ligne.

Exemple :

```
write("Ca marche"),nl
```

```
writef(Chaîne de formattage,arg1,arg2,arg3,...)
```
produit une sortie formatée. Les principaux paramètres sont :
- f pour des réels
- e pour une notation exponentielle

Exemple :

```
goal
   A= 123.456,
   writef("%3.2f",A),nl,
   writef("%10.1f",A),nl,
   writef("%e",A).
```

Affichera :

```
   123.46
       123.5
   1.23456E+02
```

Pour plus d'informations, consulter le manuel du fabricant. Nous sommes maintenant en mesure de rendre le programme plus convivial :

```
predicates
    possède(symbol,symbol)
    afficher
```

```
clauses
    possède("Jean",voiture).
    possède("Jean",livre).
    possède("Pierre",logement).
    possède("Nicole",livre).
    afficher:-
        possède(X,Y),
        write(X," possède un(e) : ",Y),nl.
```

Lançons le programme et entrons le but :

```
afficher.
```

L'écran affiche uniquement :

```
Jean possède un(e) : voiture
```

au lieu de l'ensemble des données escomptées. En effet, dès qu'un but interne est atteint, Turbo Prolog n'effectue pas de remontée à la recherche d'autres solutions. Nous verrons ultérieurement comment forcer cette remontée. Le programme peut être rendu plus convivial encore en ajoutant une section :

```
goal
    afficher.
```

Il est possible également de mettre dans la section goal, une suite de buts, le programme précédent pourrait s'écrire, par exemple :

```
predicates
    possède(symbol,symbol)

goal
    possède(X,Y),
    write(X," possède un(e) :",Y),nl.

clauses
    possède("Jean",voiture).
    possède("Jean",livre).
    possède("Pierre",logement).
    possède("Nicole",livre).
```

Lecture

Tous les prédéfinis de lecture comportent un paramètre unique : une variable obligatoirement libre au moment de l'appel. Une lecture unifie la valeur tapée au clavier à la variable paramètre. Chaque prédéfini de lecture correspond à un type donné de variable. Si l'entrée n'est pas du type requis, le prédéfini échoue.

Prédéfinis	Type du paramètre
`readln(X)`	`symbol ou string(*)`
`readint(X)`	`integer`
`readreal(X)`	`real`
`readchar(X)`	`char`

(*) La longueur d'une ligne est au maximum de 150 caractères pour les symboles et de 64K pour les chaînes.

Exemple :

```
predicates
    debut

goal
    debut.

clauses
    debut :-
        write("Donner votre nom "),nl,
        readln(Nom),nl,
        write("Donner votre age "),nl,
        readint("Age),nl,
        write("Nom : ",Nom, " Age : ",Age).
```

2.6.3. Gestion des fenêtres

Turbo Prolog possède 6 prédéfinis de gestion de fenêtres :

`makewindow(numéro_fenêtre,attribut_écran,attribut_cadre, en_tête_cadre,ligne,colonne,hauteur,largeur)`
crée une fenêtre. Les variables doivent toutes être liées.
`numéro_fenêtre`
attribue un numéro à une fenêtre.
`attribut_écran`
définit l'attribut d'affichage de l'écran. Si `attribut_cadre` est différent de zéro, un cadre entoure la fenêtre. Le haut du cadre contient `en_tête_cadre` (son titre).
`ligne` et `colonne`
spécifient les coordonnées du coin supérieur gauche de la fenêtre.
`hauteur` et `largeur`
spécifient la taille de la fenêtre.

`removewindow`
fait disparaître la fenêtre active.

`clearwindow`
efface la fenêtre active.

`shiftwindow(Numéro)`
`Numéro` est une variable entière. Si `Numéro` est libre, `shiftwindow` retourne dans `Numéro` le numéro de la fenêtre active. Si `Numéro` est liée, shiftwindow active la fenêtre correspondant à Numéro.

`window_attr(attribut)`
permet de changer l'attribut d'écran d'une fenêtre défini dans makewindow. attribut est obligatoirement lié et entier.

`window_str(Chaîne)`
Si `Chaîne` est liée, `window_str` affichera la `chaîne` dans la fenêtre. Si `Chaîne` est libre, le contenu total de la fenêtre est lié à `Chaîne`.

Pour bloquer l'affichage de l'écran tant que l'utilisateur n'a pas enfoncé une touche, utiliser : `readchar(_)`

Exemple :

```
goal
    clearwindow,
    write("Bonjour"),
    readchar(_),
    clearwindow.
```

`Bonjour` restera affiché tant que l'utilisateur n'aura pas enfoncé de touche.

2.6.4. Gestion du curseur

`cursor(Ligne,Colonne)`
Si `Ligne` et `Colonne` sont liés, le curseur passe à la position spécifié. Si `Ligne` et `Colonne` sont libres, ces paramètres sont liés aux coordonnées du curseur. Les paramètres sont entiers.

`cursorform(première_ligne,dernière_ligne)`
fixe la hauteur du curseur. Les paramètres sont entiers et obligatoirement liés et compris entre 1 et 14.

2.7. UTILISATION ALGORRITHMIQUE DU TURBO PROLOG

Comme vous avez pu le constater, Turbo Prolog est un langage fondamentalement différent des autres langages de programmation. Néanmoins, il est possible d'utiliser Turbo Prolog de manière traditionnelle. C'est à dire, d'utiliser une règle comme une procédure, de simuler alternative, aiguillage multiple ou itération. Pour simuler l'itération, voir le chapitre "Récursivité".

2.7.1. Utiliser une règle comme une procédure

Utiliser une règle comme une procédure ne présente aucune difficulté. L'en-tête de la clause correspond à l'appel de la procédure, les prémisses à une suite de sous-buts devant toujours réussir (entrées/sorties, mise en page, suite de calculs etc..). Le passage de paramètres se faisant dans l'en-tête de la règle.

Exemple :

```
predicates
  debut

goal
  debut.

clauses
  debut:-
    makewindow(1,7,7,"PROLOG",0,0,25,80),
    cursor(10,20),
    write(" Programmer en Prolog est passionnant ! "),
    readchar(_).
```

2.7.2. Simuler une structure alternative

Un test de la forme :
SI condition ALORS action I
SINON action II
sera simulé très aisément en Prolog.
Il suffit de créer un prédicat illustré par deux règles :
- la première règle comprenant les conditions du test comme premiers sous buts, suivies de la liste des actions (c'est à dire des prédicats réussissant toujours) à exécuter si la condition est vérifiée.
- la seconde règle comprenant la liste des actions à exécuter si la condition n'est pas vérifiée.

Exemple:

```
predicates
   action(integer)

goal
  write("Donner  un nombre entier : "),
  readint(X),
  action(X).
```

```
clauses
   action(X) :-
        X > 4,
        X < 8,
        write(" 4 < X < 8 ").
   action(_) :-
        write(" X > 4 ou X > 8 ").
```

Si 4 < X < 8 le sous but suivant : `write(" 4 < X < 8 ")` est atteint.

Dans le cas contraire, le sous but 4 < X < 8 n'étant pas satisfait, l'exécution se poursuit à la clause suivante `action(_)` qui réussira toujours.

2.7.3. Simuler un aiguillage multiple

Il s'agit cette fois de simuler le "case" bien connu des programmeurs traditionnels. Il suffit de créer un prédicat auquel on fera correspondre une règle pour chaque cas.

Exemple :

Le programme suivant affiche un message correspondant à chaque jour de la semaine.

```
predicates
 jour(symbol)

goal
    write(" Donner le jour de la semaine "),
    readln(X),
    jour(X).

clauses
    jour(lundi):-
        write("Hélas!").
    jour(mardi):-
        write("Courage").
    jour(mercredi):-
        write("Ca avance").
    jour(jeudi):-
        write("Il y a de l'espoir").
    jour(vendredi):-
        write("C'est presque fini").
    jour(samedi):-
        write("Bon week-end").
    jour(dimanche):-
        write("Repos").
    jour(_):-
        write("Erreur").
```

Bien entendu, il est possible d'introduire des sous buts dans chaque clause. le programme ci-dessus pourrait être écrit de la manière suivante :

```
predicates
 jour(symbol)

goal
    write(" Donner le jour de la semaine "),
    readln(X),
    jour(X).

clauses
    jour(X):-
        X=lundi,write("Hélas!").
    jour(X):-
        X= mardi,write("Courage").
    jour(X):-
        X=mercredi,write("Ca avance").
    jour(X):-
        X=jeudi,write("Il y a de l'espoir").
    jour(X):-
        X=vendredi,write("C'est presque fini").
    jour(X):-
        X=samedi,write("Bon week-end").
    jour(X):-
        X=dimanche,write("Repos").
    jour(_):-
        write("Erreur").
```

2.8. RECURSIVITE

2.8.1. Qu'est-ce-que la récursivité ?

La récursivité est la possibilité pour un prédicat de se rappeler lui-même. C'est un outil puissant que l'on emploie constamment en Turbo Prolog. Son utilisation permet de simuler des structures itératives (c'est à dire des boucles conditionnelles). Elle est également utilisée pour le traitement des listes (voir "Un type particulier d'objets complexes : les listes"). La récursivité permet aussi de définir un type d'objet complexe par rapport à lui-même (voir "Les objets complexes").

Exemple :

```
predicates
    afficher
```

```
clauses
    afficher:-
        write("Bonjour"),
    afficher.
```

Ce programme bouclera indéfiniment. Arrêtez-le en enfonçant simultanément Ctrl et Break.
L'utilisation de la récursivité implique obligatoirement un test d'arrêt.

2.8.2. Utilisation itérative de la récursivité

Les deux programmes suivants saisissent une suite de mots et s'arrêtent lorsque l'utilisateur enfonce <—┘

a) Simulation d'une structure de la forme :
REPETE traitement TANT QUE condition vraie.
Le traitement a lieu tant que la condition est vérifiée.

```
predicates
    saisie

goal
    write("Enfoncer <—┘pour quitter la saisie "),nl,
    write(),
    saisie.

clauses
    saisie:-
        write("Donnez un mot :"),nl,
        readln(X),
        X <> "",
        saisie.
```

b) Simulation d'une structure de la forme :
REPETE traitement JUSQU'A condition vraie.
Le traitement a lieu aussi longtemps que la condition n'est pas vérifiée.

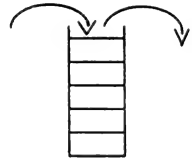
```
predicates
    saisie

goal
    write("Enfoncer <—┘pour quitter la saisie "),nl,
    write(),
    saisie.

clauses
    saisie:-
        write("Donnez un mot :"),nl,
        readln(X),
        X = "".
    saisie:-
        saisie.
```

2.8.3. Gestion des paramètres lors d'appel récursif d'un prédicat

Si la récursivité en Prolog permet à un prédicat de se rappeler lui-même, elle possède en outre la propriété de mémoriser tout le contexte à chaque appel : Prolog gère une pile pour chaque paramètre.

Mécanisme de pile

La pile se comporte exactement comme la pile de documents qui s'entasse sur le bureau du chef ; *le premier document lu est le dernier qui a été déposé* : on l'appelle le sommet de pile. Seules deux opérations sont possibles : déposer un document (il devient alors le sommet de pile, cela s'appelle empiler) ou lire un document (on suppose alors qu'il est jeté, permettant au document suivant de devenir le nouveau sommet de pile, cela s'appelle dépiler). C'est ce que l'on appelle en anglais une structure LIFO (Last In First Out)



C'est exactement ainsi que fonctionne Turbo Prolog : à chaque nouvel appel récursif d'un prédicat les valeurs des paramètres sont empilés. Au retour les paramètres sont dépilés, ils apparaissent alors, bien évidemment, dans l'ordre inverse à celui de leur stockage dans la pile.

Le programme suivant saisit une suite de noms tapés au clavier par l'utilisateur comme dans les programmes précédents, mais le `write(X)` qui suit l'appel récursif du prédicat `saisie` provoque l'affichage des noms dans l'ordre inverse de celui de la saisie.

Cet exemple illustre une autre méthode de programmation d'une boucle conditionnelle de la forme : RÉPÉTE traitement TANT QUE condition vraie.

```
predicates
  saisie(symbol)

goal
  write("Enfoncer <" pour quitter la saisie "),nl,
  write(),
  saisie(" ").

clauses
  saisie("").
  saisie(_):-
    write("Donnez un nom : "),
    readln(X),
    saisie(X),
    write(X),nl.
```

Remarque :

Si la récursivité est en fin de règle, c'est à dire que l'appel récursif du prédicat est le dernier but de la règle ou si le prédicat n'est pas paramétré, Prolog alors ne mémorise pas le contexte

des appels précédents : la récursivité est infinie. C'est ce qu'en anglais, on appelle Tail Recursion. Vous verrez au chapitre "Calculer" que l'on peut saturer la pile, dans le cas où le nombre d'appels n'est pas limité, que le prédicat n'est pas le dernier but de la règle et que celui-ci est paramétré. Vous trouverez de nombreux exemples de programmes utilisant la récursivité dans les prochains chapitres.

2.9. COMMENT FORCER OU EMPECHER LA REMONTEE ?

2.9.1. Forcer la remontée : le prédéfini fail

`Fail` signifie échec. Ce prédicat échoue toujours, il aura donc pour effet de forcer la remontée à la recherche d'autres solutions, là où Turbo Prolog le rencontre. Ajoutons `fail` à la fin de la clause `afficher` du premier programme du chapitre "Rendre un programme convivial" et relançons le programme en entrant le but `afficher`.

```
predicates
   possède(symbol,symbol)
   afficher

clauses
   possède("Jean",voiture).
   possède("Jean",livre).
   possède("Pierre",logement).
   possède("Nicole",livre).

    afficher:-
        possède(X,Y),
        write(X," possède un(e) : ",Y),
        nl,fail.
```

Cette fois, toutes les solutions s'affichent. Nous constatons toutefois qu'après l'affichage des solutions, il apparait un `No` dans la fenêtre de dialogue. Cela est tout à fait normal puisque le programme s'est terminé par `fail`. Néanmoins, si le programme devait se poursuivre il serait fâcheux de terminer par `fail`. Pour forcer l'état `Yes`, il suffit d'ajouter une clause toujours vérifiée. Or une règle sans argument ni prémisses est toujours considérée comme vraie par Turbo Prolog. C'est pourquoi nous ajouterons après la première clause `afficher` une seconde clause `afficher` sans prémisses et le programme devient :

```
predicates
   possède(symbol,symbol)
   afficher
```

```
clauses
   possède("Jean",voiture).
   possède("Jean",livre).
   possède("Pierre",logement).
   possède("Nicole",livre).

    afficher:-
        possède(X,Y),
        write(X," possède un(e) : ",Y),
        nl,fail.
    afficher.
```

Cette fois toutes les solutions sont affichées et l'exécution du programme se termine par `Yes`.

2.9.2. Empêcher la remontée : le prédéfini cut

Cut signifie coupure de choix. Son effet est l'inverse de celui de fail : il interdit la remontée. Cut s'écrie point d'exclamation : "!". Là où le programme rencontre un cut, la remontée est bloquée pour tout ce qui figure en avant du cut (c'est à dire à gauche du point d'exclamation "!") dans la règle où il figure, aussi bien pour les sous buts que pour l'en-tête de la règle elle-même. C'est à dire que *la remontée n'aura pas lieu pour les clauses suivantes du prédicat où figure le cut.*
Le cut a plusieurs usages, on l'emploie notamment :

- pour éliminer des solutions multiples en bloquant la remontée vers un ou plusieurs sous buts dans une règle.
- lorsque l'on sait qu'une solution a été trouvée et qu'il ne peut en exister d'autres. C'est alors perdre du temps et de l'espace mémoire que de remonter chercher des solutions inexistantes.

1) Eliminer des solutions multiples

Soit, par exemple, une règle r représentée par une suite de buts a, b, c, de la forme :
`r :- a,b,!,c.`
La position du cut dans cette règle bloque la remontée vers les sous buts a et b. Le programme ne trouvera que la première solution vérifiant les sous buts a et b mais toutes les solutions vérifiant le sous but c.

Exemple :

```
predicates
     aime(symbol,symbol)
     travaille(symbol,symbol,integer)
     parle(symbol,symbol)
     afficher
     terminer
```

```
goal
    afficher,
    terminer.

clauses
    aime(jean,nager).
    aime(jean,dessiner).
    aime(pierre,chanter).
    aime(louise,danser).

    travaille(jean,l_informatique,5).
    travaille(jean,la_physique,3).
    travaille(pierre,la_chimie,4).
    travaille(louise,les_mathématiques,5).

    parle(jean,anglais).
    parle(jean,espagnol).
    parle(pierre,anglais).
    parle(louise,chinois).

    afficher:-
        aime(X,Y),
        write(X," aime ",Y),nl,
        travaille(X,Z,H),
        write(X," travaille ",Z," pendant ", H," heures"),
        nl,!,parle(X,L),
        write(X," parle ",L),nl,fail.

    terminer.
```

Le programme affichera la première solution correspondant aux sous buts `aime(X,Y)` et `travaille(X,Z,H)` et toutes les solutions du sous but `parle(X,L)`.
Compte tenu de la base de faits dont nous disposons, nous apprendrons seulement que :

```
jean aime nager
jean travaille l'informatique pendant 5 heures
```

Mais en revanche nous saurons que :

```
jean parle anglais
```

et que :

```
jean parle espagnol
```

Remarquer le prédicat `terminer` obligatoire pour éliminer le `No` à la fin du programme. Une clause `afficher` sans prémisses ne serait pas testée par le programme du fait du cut.

Exemple :

Anne cherche un mari : elle désire épouser un médecin non fumeur et sportif, si elle ne trouve pas l'oiseau rare, elle se contentera d'un médecin aimant les voyages, enfin en désespoir de

cause elle acceptera un mari aimant les chevaux. Le programme suivant affichera le nom d'un mari possible par catégorie : si un ou plusieurs individus correspondent au choix préférentiel de notre héroïne, le nom du premier individu sera affichée, les choix subsidiaires ne seront pas recherchés. Au contraire si aucun individu ne répond aux premiers critères, le programme passera aux seconds critères puis aux troisièmes, s'il y a lieu.

```
predicates
    médecin(symbol)
    non_fumeur(symbol)
    sportif(symbol)
    mari(symbol)
    aime(symbol,symbol)

goal
    write("Maris possibles pour Anne :"),nl,
    mari(X),
    write(X),nl,fail.

clauses
    médecin(jules).
    médecin(théodule).
    médecin(edmond).
    médecin(joseph).

    non_fumeur(théodule).
    non_fumeur(edmond).
    non_fumeur(joseph).
    non_fumeur(pierre).

    sportif(théodule).
    sportif(joseph).
    sportif(wladimir).

    aime(jules,voyages).
    aime(théodule,voyages).
    aime(pierre,chevaux).
    aime(joseph,chevaux).

    mari(X):-
        médecin(X),
        non_fumeur(X),
        sportif(X),!.
    mari(X):-
        médecin(X),
        aime(X,voyages),!.
    mari(X):-
        aime(X,chevaux),!.
```

Seul `théodule` sera affiché, bien que `joseph` réponde également aux premiers critères (`médecin, non_fumeur et sportif`), le cut interdit son affichage. Modifier la base de faits pour vérifier les choix faits par le programme.

Remarque :

Des cuts placés au début des règles, dans le programme précédent, seraient dénués de sens. En effet, seule la première clause serait examinée. Le programme afficherait toutes les solutions correspondant au premier choix (s'il en existe) et ne passerait jamais aux deux clauses suivantes, même si il y a eu échec pour la première règle : le cut bloque la remontée à la règle suivante.

2) Eliminer les solutions impossibles

Reprenons le dernier programme du chapitre "Rendre un programme convivial" qui affiche un message correspondant au jour de la semaine entré par l'utilisateur. Nous désirons perfectionner ce programme de telle sorte qu'il redemande à l'utilisateur le jour de la semaine aussi longtemps que celui-ci n'aura pas répondu : `terminer`.

```
predicates
    debut
    jour(symbol)

goal
    debut,
    clearwindow,
    write(" Donner le jour de la semaine "),nl,
    readln(X),
    jour(X),
    X= "terminer"./*Ce sous but force la remontée*/

clauses
    jour(lundi):-
        !,write("Hélas!"),
        readchar(_)./*readchar(_) bloque l'affichage tant que
        l'utilisateur n'a pas enfoncer une touche*/
    jour(mardi):-
        !,write("Courage"),readchar(_).
    jour(mercredi):-
        !,write("Ca avance"),readchar(_).
    jour(jeudi):-
        !,write("Il y a de l'espoir"),readchar(_).
    jour(vendredi):-
        !,write("C'est presque fini"),readchar(_).
    jour(samedi):-
        !,write("Bon week-end"),readchar(_).
    jour(dimanche):-
        !,write("Repos"),readchar(_).
```

```
jour(X):-
    X<>"terminer",write("Erreur"),readchar(_).
jour(_).

debut.
debut:-debut./*Appel récursif de debut*/
```

L'usage des cuts dans ce programme est double : il évite une remontée inutile à la recherche d'autres solutions d'une part et l'unification avec l'avant dernière règle `jour(X)` qui réussirait toujours ce qui provoquerait à chaque interrogation l'impression du message : `Erreur`. Du fait de la remontée forcée par le but `X = terminer`, Turbo Prolog testerait inutilement toutes les règles, avec l'introduction des cuts, le programme devient plus performant. Pour bien comprendre ce programme et l'intérêt des cuts, suivez le pas à pas, avec et sans cuts, en activant une des directives `trace` ou `shorttrace`.

3) Les cuts inutiles

L'expérience montre que bien souvent les utilisateurs les plus avertis de Turbo Prolog placent dans leur programme des cuts parfaitement inutiles. En effet, l'utilisation du cut pour éliminer des solutions impossibles n'est nécessaire que si par ailleurs, dans le programme, l'utilisateur force cette remontée. On oublie trop souvent que Turbo Prolog s'arrête dès qu'un but est atteint. C'est ainsi que les cuts placés dans le programme suivant (similaire, aux cuts près, à celui du chapitre "Rendre un programme convivial") sont totalement inutiles. Pour vous en convaincre, suivez les deux programmes (avec et sans cuts) en pas à pas et vous constaterez que dans l'un et l'autre cas, dès qu'une règle a été satisfaite, Turbo Prolog ne poursuit pas sa recherche : les cuts n'ajoutent rien au programme.

```
predicates
    jour(symbol)

goal
    write(" Donner le jour de la semaine "),
    readln(X),
    jour(X).

clauses
    jour(lundi):-
        !,write("Hélas!").
    jour(mardi):-
        !,write("Courage").
    jour(mercredi):-
        !,write("Ca avance").
    jour(jeudi):-
        !,write("Il y a de l'espoir").
    jour(vendredi):-
        !,write("C'est presque fini").
```

```
    jour(samedi):-
        !,write("Bon week-end").
    jour(dimanche):-
        !,write("Repos").
    jour(_):-
        !,write("Erreur").
```

Le but du programme donné au début de cette section (Eliminer les solutions impossibles) est de faire comprendre au lecteur dans quel cas est-ce_que l'on peut être amené à utiliser le cut pour éliminer des solutions impossibles. En fait, la solution donnée ci-dessous est la mieux programmée.

```
predicates
    debut
    jour(symbol)

goal
    debut.

clauses
    debut:-
        clearwindow,
        write(" Donner le jour de la semaine "),nl,
        readln(X),jour(X),
        X<> "terminer",!,
        debut./*Appel récursif de debut*/

    jour(lundi):-
        write("Hélas!"),readchar(_).
    jour(mardi):-
        write("Courage"),readchar(_).
    jour(mercredi):-
        write("Ca avance"),readchar(_).
    jour(jeudi):-
        write("Il y a de l'espoir"),readchar(_).
    jour(vendredi):-
        write("C'est presque fini"),readchar(_).
    jour(samedi):-
        write("Bon week-end"),readchar(_).
    jour(dimanche):-
        write("Repos"),readchar(_).
    jour(X):-
        X<>"terminer",write("Erreur"),readchar(_).
    jour(_).
```

L'unique cut du programme permet son arrêt, en bloquant la remontée vers `debut` lorsque l'utilisateur a tapé : `terminer`. Ici encore des cuts dans chaque règle sont inutiles.

2.9.3. Négation d'un prédicat

Le prédéfini not permet d'exprimer la négation d'un prédicat.

Exemple :

```
predicates
     fumeur(symbol)
     sportif(symbol)
     afficher

goal
    afficher.

clauses
     fumeur(jules).
     fumeur(joseph).

     sportif(jules).
     sportif(pierre).
     sportif(jean).

     afficher:-
        write("Liste des sportifs non fumeurs :"),
        nl,sportif(X),
        not(fumeur(X)),write(X),nl,fail.
     afficher.
```

Pour Turbo Prolog tout ce qui n'est pas expressément déclaré vrai est réputé faux. Pierre et jean n'étant pas déclarés fumeurs, il est vrai (au sens Turbo Prolog) qu'ils sont non fumeurs. On peut obtenir la négation d'un prédicat sans avoir usage au prédéfini not.

```
predicates
    fumeur(symbol)
    sportif(symbol)
    non_fumeur(symbol)
    afficher

goal
   afficher.

clauses
    non_fumeur(X):-
        fumeur(X),!,fail.
    non_fumeur(_).

    fumeur(jules).
    fumeur(joseph).
```

```
sportif(jules).
sportif(pierre).
sportif(jean).

afficher:-
    write("Liste des sportifs non fumeurs :"),
    nl,sportif(X),
    non_fumeur(X),
    write(X),nl,fail.
afficher.
```

Le prédicat `nom_fumeur` correspond à `not(fumeur)`, Turbo Prolog ne s'y prend pas autrement pour créer la négation d'un prédicat. Ce que l'on peut écrire :

```
non_p :- p,!,fail.
non_p.
```

Si `p` réussit on lui crée l'impasse : `!,fail`. Ce qui bloque la remontée et fait échouer `non_p`. Si `p` échoue, la seconde clause `non_p` (sans prémisses réussira).

2.10. OBJETS COMPLEXES

2.10.1. Qu'est ce qu'un objet complexe ?

Appelé également objet composé, un objet complexe est un objet composé d'autres objets. Un objet complexe est traité par Prolog comme un objet unique, ce qui simplifie les clauses.
La déclaration d'objets complexes permet de rassembler les objets par catégorie.

2.10.2. Déclaration d'objets complexes

Reprenons le programme du chapitre "Rendre un programme convivial" qui exprime que des individus possèdent quelque chose. Si nous voulons préciser qu'une voiture a non seulement une marque mais également une couleur nous déclarerons un foncteur `voiture` :

```
domains
   voiture = voiture(marque,couleur)
   nom,marque,couleur = symbol

predicates
   possède(nom,voiture)

clauses
   possède("Jean",voiture(renault_5,bleue)).
```

La section domains peut également se définir sous la forme :

```
domains
   voiture = voiture(symbol,symbol)
```

Il est important de ne pas faire de confusion entre foncteur et prédicat : un prédicat retourne une valeur vraie ou fausse en fonction de ses arguments, *un foncteur est un objet constitué d'autres objets plus élémentaires*. Tel que nos objets sont conçus pour le moment, nous ne pourrons relier le prédicat possède qu'à une voiture. Turbo Prolog permet de regrouper des objets de nature diverse sous un même type.

```
domains
 chose =
   voiture(marque,couleur);
   logement(nature,nb_étages);
   livre(auteur,titre);
   bateau

 marque,couleur,nature,
 auteur,titre = symbol

 nb_étages = integer
```

Le type chose désigne une voiture OU un logement OU un livre OU un bateau. Voiture, logement, livre et bateau sont des foncteurs. Noter qu'un foncteur n'a pas obligatoirement d'arguments (tout comme les prédicats). C'est le cas, dans notre exemple, du foncteur bateau. Noter le point virgule ";" qui signifie OU. La déclaration du type chose peut également se faire sous la forme simplifiée :

```
domains
 chose =
   voiture(symbol,symbol);
   logement(symbol,integer);
   livre(symbol,symbol);
   bateau
```

Un objet complexe peut lui même contenir des objets complexes. Ainsi, si nous voulons exprimer qu'un livre est composé d'un titre et d'un auteur et qu'un auteur est lui même composé d'un nom et d'un prénom, nous déclarerons :

```
domains
 chose =
   voiture(marque,couleur);
   logement(nature,nb_étages);
   livre(auteur,titre);
   bateau
```

```
 marque,couleur,nature,
 titre,nom,prénom = symbol

 nb_étages = integer

 auteur = auteur(nom,prénom)
```

Ou la forme simplifiée :

```
domains
 chose =
   voiture(symbol,symbol);
   logement(symbol,integer);
   livre(auteur,symbol);
   bateau

 auteur = auteur(symbol,symbol)
```

Exemple :

```
domains
 chose =
   voiture(symbol,symbol);
   logement(symbol,integer);
   livre(auteur,symbol);
   bateau

 auteur = auteur(symbol,symbol)

predicates
   possède(symbol,chose)

clauses
   possède("Jean",voiture("Renault 5","bleue")).
   possède("Jean",livre(auteur("Castelot","André"),
   "L'aiglon")).
   possède("Pierre",logement(chateau,3)).
   possède("Nicole",livre(auteur("Lapierre",
   "Dominique"),"La cité de la joie")).
   possède("Catherine",bateau).
```

Formuler les buts :
Qui possède quoi ?
Soit : `possède(Qui,Quoi)`
Qui possède un livre ?
Soit : `possède(Qui,livre(auteur(_,_),_))`

2.10.3. Comment tester l'égalité de deux objets Prolog ?

Pour tester l'égalité de deux objets Prolog, il suffit de définir le prédicat `égal(objet,objet)` suivant. Ce prédicat retourne Yes si les deux paramètres sont identiques. Les objets peuvent être de tous types : integer, symbol, composés, listes etc..

```
predicates
 égal(objet,objet)
clauses
 égal(X,X).
```

Il faut, bien entendu pour utiliser ce prédicat, déclarer `objet` dans la section `domains`.

2.10.4. Définition récursive d'objets complexes

Nous développerons peu cette notion. En effet, définir un objet récursivement revient à faire soi-même le travail de Turbo Prolog lorsque nous lui demandons de créer une liste. Aussi, nous vous proposons d'être intelligemment paresseux et d'utiliser les listes plutôt que les objets complexes définis récursivement. Un objet défini récursivement est déclaré dans la section `domains` par deux états possibles : l'état non vide OU (;) l'état vide (critère d'arrêt). Ce qui s'écrira en Turbo Prolog :

```
domains
liste_objets =
    liste(objets,liste_objets);
    vide
objets = ..
```

`liste_objets` est un objet complexe, qui est :
- soit le foncteur `liste`, c'est à dire une liste non vide.
- soit le foncteur sans argument `vide`, c'est à dire une liste vide. `objets` désigne un élément de `liste` dont le type est déclaré dans la section `domains` et `liste_objets` les autres objets de la liste. Turbo Prolog ne fait rien d'autre que ce que nous venons de faire lorsque nous créons une liste d'objets (voir Chapitre suivant).

2.11. UN TYPE PARTICULIER D'OBJETS COMPLEXES : LES LISTES

2.11.1. Définition

Une liste est une suite d'objets Prolog de type identique. Une liste est délimitée par des crochets et ses éléments sont séparés par des virgules. Une liste peut, dans une certaine mesure, être comparée à un tableau d'un langage traditionnel. Il existe cependant une différence fondamentale entre un tableau et une liste : on accède, dans un tableau, directement à l'élément de rang "n", alors que le parcours d'une liste est séquentiel, élément après élément.

2.11.2. Déclaration d'une liste

Une liste est obligatoirement déclarée dans la section `domains`, sous la forme :

```
domains
 liste_objets = objets*
  objets = ...
```

L'étoile "*" spécifie une liste. On peut constituer des listes de tout type d'objets : entiers, symboles etc.. y compris des listes de listes, à la seule condition que tous les objets de la liste soient de même type.

Exemple de déclaration :

```
domains
   liste_char = char* /*liste de caractères*/
   liste_symbol = symbol* /*liste de symboles*/
   liste_de_liste = liste_symbol* /*liste de listes de
 symboles*/
```

Exemples de liste :

`['a','b','c','d']`	est une liste de caractères.
`[veau,vache,cochon]`	est une liste de symboles.
`[[vert,rouge],[bleu,blanc,rouge],[blanc]]`	est une liste de listes de symboles.
`[]`	est une liste vide

2.11.3. Autre définition des listes

Les listes sont importantes en Turbo Prolog et pour les manipuler efficacement il existe un autre mode de représentation : la segmentation de la liste en deux parties : la tête et la queue.
- La tête est le premier élément de la liste.
- La queue est elle même une liste composée de tous les éléments suivants de la liste. La tête et la queue sont séparées par une barre verticale tronquée (code ASCII 124).

Exemples d'unification entre liste et valeurs :

```
domains
 liste_symbol = symbol*
predicates
 liste(liste_symbol)
clauses
 liste([bleu,blanc,rouge]).

Goal: liste(X)
X = [bleu,blanc,rouge]
1 Solution
```

```
Goal: liste([X,Y,Z]) X= bleu, Y= blanc, Z= rouge
1 Solution

Goal: liste([T¦Q])
T = bleu et Q = [blanc,rouge]
1 Solution

Goal: liste([T¦[X,Y])
T = bleu, X = blanc, Y =rouge
1 Solution
Goal:liste([X,Y,Z,T]) No solution
```

L'unification a échouée.

2.11.4. Exemples de manipulation de liste

Les exemples qui suivent correspondent à des prédicats courants et bien utiles pour manipuler les listes. Pour tester les divers exemples, il vous suffira de définir dans la section `domains` la liste et le type de ses objets par une déclaration :

```
domains
   liste  = objets*
   objets = ..
```

Test de liste vide

Le prédicat `vide(liste)` retourne `Yes`, si `liste` est vide, `No` dans le cas contraire.

```
predicates
   vide(liste)
clauses
   vide([]).
```

Ajout d'un élément en début de liste

Le prédicat `ajout_début(x,liste1,Liste2)` forme `Liste2` qui a pour tête `x` et pour queue `liste1`.

```
predicates
   ajout_début(objet,liste,liste)
clauses
   ajout_début(X,L1,[X|L1]).
```

Pour tester l'égalité de deux listes voir au chapitre précédent : l'égalité de deux objets Prolog. Les prédicats qui suivent sont définis récursivement. La queue d'une liste étant elle-même une liste, le principe du raisonnement est toujours le même, à savoir :
- l'arrêt du processus lorsque la solution cherchée a été obtenue ou lorsque la liste passée comme paramètre est vide.
- l'appel récursif du prédicat sur la queue de la liste.

Afficher les éléments d'une liste

Le prédicat `affiche(liste)` affichera tous les éléments de liste.

```
predicates
 affiche(liste)
clauses
 affiche([]).*Arrêt si la liste est vide*/
 affiche([T|Q]):- write(T),nl,
   affiche(Q).*Appel récursif sur la queue de la liste*/
```

Appartenance à une liste

Le prédicat élément(x,liste) retourne Yes si x est élément de liste, No dans le cas contraire.

```
predicates
 élément(objets,liste)
clauses
 élément(X,[X|_]).*arrêt si X est identique à la tête de la
liste*/
 élément(X,[_|Q]):-élément(X,Q).
```

Concatèner deux listes

Le prédicat concatène `(liste1,liste2,Liste3)` concatène `liste1` et `liste2` pour former `Liste3`.

```
predicates
 concaténer(liste,liste,liste)
clauses
 concaténer([T1|Q1],L2,[T1|Q3]):-
   concaténer(Q1,L2,Q3).
```

Ajout d'un élément en fin de liste

Le prédicat `ajout_fin(x,liste1,Liste2)` forme `Liste2` obtenue en ajoutant `x` à la fin de `liste1`. Attention `ajout_fin` appelle `concaténer`.

```
predicates
 ajout_fin(objets,liste,liste)
clauses
 ajout_fin(X,L1,L2):-
   concaténer(L1,[X],L2).
```

Inverser une liste

Le prédicat `inverse(liste1,Liste2)` forme `Liste2` qui est la liste inversée de `liste1`.

```
predicates
 renverse(liste,liste,liste)
 inverse(liste,liste)
clauses
 inverse(L1,L3):- renverse(L1,[],L3).
 renverse([],L3,L3)./*Arrêt si L1 est vide*/
 renverse([T1|Q1],L2,L3):-
 renverse(Q1,[T1|L2],L3).
```

Retourner le dernier élément d'une liste

Le prédicat `dernier(X,liste)` retourne dans `X` le dernier élément de `liste`.

```
predicates
 dernier(objets,liste)
clauses
 dernier(T,[T|[]]).
 dernier(T,[_|Q]):-
dernier(T,Q).
```

Construire une liste à partir d'objets saisis au clavier

```
domains
 liste_symbol = symbol*
predicates
 ajout(liste_symbol,liste_symbol)
clauses
clauses
 ajout([""|L],L):-!.
 ajout(L2,L3):-
   readln(X),
 ajout([X|L2],L3).
```

Exemple de requête :

```
goal: ajout([],L)
```

La liste initiale est vide et la liste résultat est L. Contrairement aux exemples précédents le prédicat est donné pour une liste de symboles ou de chaînes. Pour construire une liste d'entiers par exemple, le test d'arrêt ("") doit être modifié ainsi que le prédicat de lecture.

2.12. CHAINES DE CARACTERES

Les chaînes de caractères sont de type `string` ou `symbol` en Turbo Prolog. Les deux types de chaînes se manipulent de manière identique. La seule différence c'est que les chaînes de type `string` sont obligatoirement encadrées de guillemets, alors que les chaînes de type `symbol` peuvent être encadrées de guillemets ou non. Dans ce dernier cas, elles ne doivent pas contenir de caractère espace et doivent obligatoirement commencées par une lettre minuscule.

2.12.1. Comment parcourir une chaîne séquentiellement ?

Les deux prédicats `frontchar` et `fronttoken` permettent de traiter récursivement les chaînes de caractères de manière analogue au traitement des listes à l'aide des entités : tête et queue. Tous les programmes de manipulation de liste du chapitre précédant peuvent être réécris pour les chaînes. Seule la syntaxe diffère, la logique des programmes est inchangée.

Prédicat frontchar

Ce prédicat permet, soit d'extraire le premier caractère d'une chaîne, soit d'ajouter un caractère en début de chaîne. Ces deux modes d'utilisation de `frontchar` correspondent à deux syntaxes différentes :

1re syntaxe

```
frontchar(chaîne,Premier_caractère,Fin_de_chaîne)
```
- `chaîne` est la chaîne à manipuler.
- `Premier_caractère` est une variable libre qui recevra le premier caractère de chaîne (c'est en quelque sorte la tête de la chaîne).
- `Fin_de_chaîne` est une variable libre qui recevra la chaîne privée de son premier caractère (c'est la queue de la chaîne).

2ème syntaxe

```
frontchar(Nouvelle_chaîne,caractère,chaîne)
```
- `Nouvelle_chaîne` est une variable libre qui recevra la nouvelle chaîne.
- `caractère` est le caractère à ajouter en début de chaîne.
- `chaîne` est la chaîne initiale à manipuler.

Exemple :

```
Goal: frontchar(bonjour,Premier,Fin)
Premier=b, Fin=onjour
1 solution
Goal: frontchar(Chaîne,'b',onjour)
Chaîne= bonjour
1 solution
```

Prédicat fronttoken

Ce prédicat permet, soit d'extraire la première sous-chaîne d'une chaîne, soit d'ajouter une sous-chaîne en début de chaîne. Une séquence de caractères constitue une sous chaîne (token) au sens de Turbo Prolog, si :
- c'est un symbol au sens de Turbo Prolog
- c'est un nombre introduit par une rupture : espace ou guillemet.

Les deux modes d'utilisation correspondent aux deux syntaxes :

1re syntaxe

`fronttoken(chaîne,Sous_chaîne,Fin_de_chaîne)`
- `chaîne` est la chaîne à manipuler.
- `Sous_chaîne` est une variable libre qui recevra la première sous-chaîne de `chaîne`.
- `Fin_de_chaîne` est une variable libre qui recevra la chaîne privée de sa première sous chaîne.

2ème syntaxe

`fronttoken(Nouvelle_chaîne,sous_chaîne,chaîne)`
- Nouvelle_chaîne est une variable libre qui recevra la nouvelle chaîne.
- `sous_chaîne` est la chaîne à ajouter en début de chaîne.
- `chaîne` est la chaîne initiale à manipuler.

Exemple :

```
Goal:fronttoken("Ca marche bien",Premier,Fin)
Premier=Ca
Fin= marche bien
1 solution
Goal:fronttoken("256élèves,Premier,Fin)
Premier=256
Fin=élèves
1 solution
Goal:fronttoken("matricule306",Premier,Fin)
Premier=matricule306
Fin=
1 solution
```

2.12.2. Transformer une chaîne en liste de caractères ou de symboles

Le programme suivant transforme une chaîne de caractères en une liste de caractères :

```
domains
 liste = char*
predicates
 liste_char(string,liste)
clauses
 liste_char(C,[T|Q]):-
   frontchar(C,T,X),!,
   /*Appel récursif de liste_char sur la queue de la chaîne*/
   liste_char(X,Q).
liste_char(_,[]).
```

Exemple de but :

```
Goal:liste_char(bonjour,X)
X=['b','o','n','j','o','u','r']
```

Le programme qui transforme une chaîne en une liste de symboles est semblable au précédant : remplacer `frontchar` par`fronttoken`.

2.12.3. Autres prédicats de traitement de chaînes

Nous n'insisterons pas beaucoup sur les autres prédicats de traitement des chaînes qui sont beaucoup plus classiques et qui correspondent à des fonctions standard en matière de traitement de chaînes dans tous les langages.

`concat(chaîne1,chaîne2,Ghaîne3)`
lie `Chaîne3` (libre) à la concaténation de `chaîne1` et `chaîne2` (liées).

`frontstr(n,chaîne1,Chaîne2,Chaîne3)`
tronque `chaîne1` (liée) en deux, `Chaîne2` (libre) reçoit les n premiers caractères et `Chaîne3` (libre) la fin.

`str_len(chaîne,Longueur)`
retourne dans `Longueur` (libre ou liée), la longueur de chaîne (liée) ou vérifie que la longueur de `chaîne` est bien `longueur`.

2.12.4. Prédéfinis de conversion

`upper_lower(Chaîne_maj,Chaîne_min)`
Ce prédicat lie les deux paramètres aux versions majuscules et minuscules de la même chaîne. Les paramètres peuvent être libres ou liés.

`str_int(Chaîne,Entier)`
Si `Chaîne` est libre, `str_int` lie `Chaîne` à la chaîne de chiffres décimaux correspondant à Entier de type integer.
Si `Entier` est libre, `str_int` lie `Entier` à la valeur entière correspondant à la chaîne numérique.
Si les deux paramètres sont liés, il y a succès si l'unification réussit.

`str_real(Chaîne,Réel)` et `str_char(Chaîne,Caractère)`
Ces deux prédicats fonctionnent de manière analogue à `str_int`, le second paramètre étant de type `real` ou `char`.

`str_char(Caractère,Code_ascii)`
lie `Caractère` à la valeur du code ASCII correspondant. Le fonctionnement est analogue aux prédicats précédents.

2.13. CALCULER

2.13.1. Généralités

Turbo Prolog possède, comme tous les langages de programmation classiques, les quatre opérateurs de base sur des entiers et des réels, les opérateurs modulo (`mod`) et division entière (`div`) ainsi qu'une gamme de fonctions mathématiques. Les règles d'évaluation d'une expression arithmétique sont analogues à celles de tous les langages :
- Les sous-expressions entre parenthèses sont évaluées en premier.
- A priorité égale l'évaluation se fait de gauche à droite.
- L'ordre de priorité des opérateurs est donné par le tableau suivant :

Opérateur	Priorité
+ -	1
* /	2
mod div	3
+ -(unaires)	4

- Si les deux opérandes d'une opération sont de même type, le type est conservé dans le résultat. Le type du résultat est réel si les opérandes sont de type différent.

2.13.2. Egalité

Turbo Prolog, ne l'oublions pas, ne raisonne qu'en terme de vrai/faux, le signe égal "=" n'échappe pas à ce principe.
Turbo Prolog utilise la notation classique :
`X = expression`
au lieu de la notation normale qui serait :
`= (X,expression)`

La notation classique, agréable à manipuler pour le programmeur, ne doit pas masquer la véritable signification du signe "=". Lorsque Turbo Prolog rencontre un signe "=", l'expression à droite du signe égal est évaluée, ce qui implique que toutes les variables situées à droite du signe "=" soient liées au moment de l'appel de cet opérateur. Puis, dans une seconde étape, Turbo Prolog essaie d'unifier le résultat de l'évaluation avec ce qui figure à gauche du signe "=".
Deux cas peuvent alors se produire :
- le membre de gauche est une variable libre, dans ce cas elle est unifiée au résultat. Le signe égal s'est comporté comme l'opérateur d'affectation d'un autre langage.
- le membre de gauche est une variable liée ou une expression contenant une variable liée, auquel cas l'unification ne peut réussir que si les deux membres sont de valeur identique. Le signe égal s'est comporté comme un opérateur de comparaison.

Exemple :

Rentrer les buts suivants :

```
Goal:X=4
X=4
1 solution
```

L'unification a réussi.

```
Goal:X=1,X=X+2
No Solution
```

L'unification échoue forcément : X étant différent de X+2. Les très classiques N = N + 1 du Basic ou N := N + 1 du Pascal n'ont aucun sens en Prolog, N étant toujours différent de N + 1. Remarquer la logique de Prolog, N = N + 1 n'a aucun sens en algèbre !

Exemple :

```
predicates
   calcul(integer,integer)
clauses
   calcul(X,Y):- X+2 = Y+3.
```

Rentrer les buts :

```
Goal: calcul(3,7)
No
Goal: calcul(4,3)
Yes
```

Attention, X et Y doivent être obligatoirement liés quand calcul est appelé.

Exemple de programme récursif :

Le programme suivant décrémente N jusqu'à 0.

```
predicates
   décrem(integer)

clauses
   décrem(0)./*Test d'arrêt, quand le paramètre = 0*/
   décrem(N):-
      N1= N-1,
      write(N1," "),
      décrem(N1).
```

Le test d'arrêt peut être inclus dans la règle décrem(N) de la manière suivante :

```
predicates
   décrem(integer)

clauses
   décrem(N):-
      N1= N-1,
      write(N1," "),
      N1 > 0,
      décrem(N1).
```

Exécuter ce programme, en rentrant le but `décrem(10)` par exemple. Permuter les deux derniers sous buts (`décrem(N1)` et `write(N1," ")` du premier programme. Vous constatez que cette fois le but `décrem(10)` affiche les nombres de 0 à 9 par ordre croissant. Suivez en mode trace l'exécution du programme pour comprendre le mécanisme de la récursivité. : le prédicat `décrem` se rappelle lui-même par `décrem(N1)`, les valeurs successives de N1 sont empilées dans leur ordre d'apparition. Quand la clause `décrem(0)` réussit, l'exécution se poursuit au sous but suivant `write(N1," ")`, les valeurs de N1 sont dépilées et affichées à chaque retour de `décrem`. Par défaut la taille de la pile est de 600 paragraphes de 16 octets, pour augmenter la taille de la pile, utiliser la directive "Memory allocation" (voir chapitre "Mise en oeuvre de Turbo Prolog").

Remarque :

Supprimez le test d'arrêt du premier programme `décrem(0)`, et faites le tourner, vous constatez alors qu'il va boucler indéfiniment dans le cas où l'appel récursif de `décrem` est le dernier sous-but. C'est la "Tail recursion" dont nous avons parlé au chapitre "Récursivité". Si vous ajoutez un sous-but `write(N1," ")` après l'appel récursif `décrem(N1)`, vous saturez rapidement la pile ("Stack overflow").

2.13.3. Fonctions mathématiques

Les fonctions mathématiques de Turbo Prolog s'utilisent comme les fonctions d'un autre langage dans un prédicat de comparaison ou d'affectation. Les paramètre x et y de ces fonctions sont obligatoirement liés.

`abs(x)`
retourne la valeur absolue de x.

`ln(x) et log(x)`
retourne les logarithmes népérien et décimaux de x.

`exp(x)`
retourne "e" à la puissance x.

`sqrt(x)`
retourne la racine carré de x

`sin(x), cos(x) et tan(x)`
retournent le sinus, le cosinus et la tangente de x exprimé en radians.

`x mod y`
retourne le reste de la division entière de x par y.

`x div y`
retourne le quotient de x par y.

Exemple :

```
Goal: X = abs(-5)
X=5
1 Solution
Goal: X= log(100)
X=2
1 Solution
```

2.13.4. Générer des nombres au hasard

`random(X)`
Ce prédicat retourne dans la variable libre X, un nombre aléatoire compris entre 0 et 1.

Exemple :

```
Goal: random(X)
X= 0.45851651011
1 Solution
```

2.14. COMMENT METTRE A JOUR UNE BASE DE CONNAISSANCES ?

Jusqu'ici nous avons créé des bases de connaissances statiques, internes au programme. Or Turbo Prolog permet de créer des bases de faits (de données) dynamiques. Une base de données dynamique est une base dans laquelle il est possible d'ajouter ou de supprimer des faits pendant l'exécution du programme. Cette base de faits est créée en mémoire vive. Il n'existe pas de prédicats permettant de créer des bases de règles dynamiques. La création d'un tel module est à la charge du programmeur. Vous trouverez au chapitre : "Un Système Générateur de Système Expert", un exemple entièrement commenté d'un tel programme.

2.14.1. Déclarer une base de données dynamique

Les prédicats destinés à constituer une base de données dynamique sont déclarés dans une section particulière du programme introduite par le mot clé `database`. La section `database` précède la section `predicates`. Les prédicats de section `database` s'utilisent comme les autres prédicats.

2.14.2. Mettre à jour une base de données dynamique

Trois prédicats permettent de mettre à jour une base de données dynamiques :

`asserta(fait)`
insère le fait spécifié en tête de la base de faits.

`assertz(fait)`
ajoute le fait spécifié en queue de la base de faits.

`retract(fait)`
retranche le fait spécifié de la base de données.

Exemple :

```
database
 habite(symbol,symbol)

goal
 asserta(habite(pierre,lille)),
/*ajoute habite(pierre,lille) à la base*/
 retract(habite(_,_).
/*détruit tous les faits relatifs au prédicat habite*/
```

Il est possible de spécifier dans les prédicats `asserta`, `assertz` et `retract` le nom d'une base dynamique (il est donc possible de gérer plusieurs bases dynamiques dans un même programme.

Exemple :

```
database - lieu
 habite(symbol,symbol)

goal
 asserta(habite(pierre,lille),lieu),
/*ajoute habite(pierre,lille) à la base lieu*/
 retract(habite(_,_),lieu).
/*détruit tous les faits relatifs au prédicat habite dans la
base lieu*/
```

2.14.3. Sauvegarder une base de données sur disque

`save("nom_fichier_dos")`
sauve la base de données dynamique dans le fichier disque spécifié.
La base de faits sera sauvée avec un fait par ligne.

`consult("nom_fichier_dos")`
lit la base de faits préalablement sauvée sur le disque sous le nom spécifié. Le prédicat consult réussit si la base de faits du fichier est sans erreur, sinon il échoue.

`deletefile("nom_fichier_dos")`
détruit le fichier du disque spécifié.

2.14.4. Exemple : gestion d'une base de faits dynamique

```
database
 habite(symbol,symbol)

predicates
 menu
 exec(char)
 vide(symbol)
 sauve(char)

goal
 menu.

clauses
 menu:-
  makewindow(1,7,7,"",0,0,25,80),
  clearwindow,
  write("Ajouter des faits (1)"),nl,
```

```
 write("Lister la base (2)"),nl,
 write("Afficher un fait (3) "),nl,
 write("Supprimer un fait (4) "),nl,
 write("Détruire la base en mémoire (5) "),nl,
 write("Sauver la base sur disque (6) "),nl,
 write("Charger une base (7) "),nl,
 write("Détruire une base sur le disque (8)"),nl,
 write("Quitter le programme (9)"),nl,
 write("Taper le numéro correspondant à votre choix "),
 nl,
 readchar(X),
 clearwindow,
 exec(X),
 menu.
 menu.

exec('1'):-
 clearwindow,
 write("Donnez le nom : "),nl,
 readln(Nom),not(vide(Nom)),
 write("Donnez la ville :"),nl,
 readln(Ville),nl,
 assertz(habite(Nom,Ville)),
 exec('1').

exec('2'):-
 habite(Nom,Ville),
 write(Nom," habite ",Ville),nl,
 readchar(_),
 fail.
 exec('2').

exec('3'):-
 write("Donner un nom :"),nl,
 readln(Nom),nl,
 clearwindow,
 habite(Nom,Ville),
 write(Nom," habite ",Ville),
 readchar(_).

exec('4'):- write("donner le nom à supprimer :"),nl,
 readln(Nom),
 retract(habite(Nom,_)).

exec('5'):-
 retract(habite(_,_)),fail.
 exec('5'):-
 write("La base a a été détruite "),
```

```
 readchar(_).

exec('6'):-
  sauve('O').

exec('7'):-
 clearwindow,
 write("Donner le nom de la base à charger :"),
 nl,readln(Base),
 consult(Base).

exec('8'):-
 write("Donner le nom de la base à détruire "),nl,
 readln(Base),
 deletefile(Base),
 write(Base,"est détruite"),
 readchar(_).

exec('9'):-
 write("Voulez-vous sauver la base (O/N) ? "),nl,
 readchar(X),
 sauve(X),
 !,retract(habite(_,_)),fail.

exec(_).

vide("")./*Pour quitter la saisie*/

sauve('O'):-
 write("Donner le nom de la base à sauver :"),
 nl,readln(Base),
 save(Base).

 sauve('N').

 sauve(_):-
   clearwindow,
   write("Répondez par O ou N "),nl,
   readchar(X),
   sauve(X).
```

2.15. AUTRES POSSIBILITES DE TURBO PROLOG

Vous trouverez brièvement décrites dans ce chapitre les possibilités standard de Turbo Prolog. C'est à dire l'ensemble des fonctions communes à tous les langages de programmation.

2.15.1. Fichiers

Un fichier doit être déclaré par un nom symbolique dans la section `domains` par :

```
file = nom_symbolique
```

Ouverture de fichier

`openread(nom_symbolique,"nom_fichier_DOS")`
Ouverture en lecture exclusivement.

`openwrite(nom_symbolique,"nom_fichier_DOS")`
Ouverture en écriture exclusivement.

`openappend(nom_symbolique,"nom_fichier_DOS")`
Ouverture en ajout en fin de fichier.

`openmodify(nom_symbolique,"nom_fichier_DOS")`
Ouverture en lecture et écriture.

Lecture et écriture d'un fichier

Celles-ci se font par les ordres de lecture et écriture normaux (`readln, readint`,... et `write`). Pour qu'une opération de lecture ou d'écriture ait lieu dans un fichier il faut après l'avoir ouvert, assigner l'unité standard de lecture ou d'écriture au fichier par :

```
readdevice(nom_symbolique)
writedevice(nom_symbolique)
```

On revient aux unités standard de lecture/écriture par :

```
readdevice(keyboard)
writedevice(screen)
```

L'imprimante n'a pas à être ouverte par `openwrite` mais uniquement réassignée par :

```
writedevice(printer)
```
Fermeture d'un fichier

`closefile(nom_symbolique)`
ferme le fichier spécifié.

`Fin de fichier`
`eof(nom_symbolique)`
réussit si la fin de fichier est atteinte

Exemple :

Ce programme saisit un nom au clavier, écrit ce nom dans le fichier disque `clients.txt`. Puis lit le nom dans le fichier disque pour l'afficher à l'écran.

```
domains
     file = fichier

predicates
      lire_écrire

goal
      lire_écrire.

clauses
     lire_écrire:-
     /* Saisie au clavier*/
     write("donner un nom"),nl,
     readln(Nom),
     /* Ecriture dans le fichier*/
     openwrite(fichier,"clients.txt"),
     writedevice(fichier),
     write(Nom),
     closefile(fichier),
     readdevice(keyboard),
     /*Lecture dans le fichier*/
     openread(fichier,"clients.txt"),
     readdevice(fichier),
     readln(N),
     /* Affichage à l'écran*/
     writedevice(screen),
     clearwindow,
     write("Nom :",N),
     closefile(fichier).
```

Après avoir exécuter ce programme, vérifiez qu'un fichier `clients.txt` a été créé et qu'il contient le nom saisi.

Gestion de fichier

`existfile("nom_fichier_DOS")`
réussit si le fichier existe dans le répertoire actif.

`deletefile("nom_fichier_DOS")`
détruit sur le disque le fichier spécifié.

`renamefile("ancien_nom_DOS,"nouveau_nom_DOS")`
analogue à la commande RENAME du DOS.

2.15.2. Graphisme

Les possibilités graphiques de la version 2.0 du logiciel ont été considérablement étendues par rapport à la version 1.0. Elles comprennent tous les prédicats graphiques de la version 1.0, augmentées de nouveaux prédicats. Si vous avez déjà l'habitude d'un langage Borland (Turbo Pascal ou Turbo C), vous retrouverez les mêmes routines graphiques et les mêmes techniques de programmation pour activer ces nouveaux prédicats.

1) Graphisme propre à la version 2.0

Activer le mode graphique

La première ligne du programme doit utiliser la directive :

```
include "grapdecl.pro"
```

qui inclut le fichier "grapdecl.pro" contenant les routines graphiques de Prolog.

Prédicats de mise en oeuvre du mode graphique 2.0

```
initgraph(detect,mode,Graph,Mode,"chemin")
```

On active le mode graphique en utilisant le prédicat `initgraph` avec les paramètres suivants :
- `detect` permet une autodetection de la carte graphique de l'ordinateur.
- `mode` est un entier compris entre 1 et 5 qui caractérise le mode graphique (le nombre de pixels par ligne et colonne)

mode	Résolution	Adaptateur	Nbr de couleurs
1	320x200	CGA	4
2	640x200	CGA	2
3	320x200	EGA	16
4	640x200	EGA	16
5	640x350	EGA	13

`Graph` et `Mode` sont deux paramètres libres à l'appel du prédicat et liés par défaut par `grapdecl.pro`.

`chemin` correspond au chemin du répertoire contenant les fichiers .BGI.

`setbkcolor(couleur)`
sélectionne les couleurs

Constantes	couleurs
black	0
blue	1
green	2
cyan	3
red	4
magenta	5
brown	6
lightgrey	7
darkgrey	8
lightblue	9
lightgreen	10
lightcyan	11
lightred	12
lightmagenta	13
yellow	14
white	15
blink	128

Ajouter blink à la couleur pour faire clignoter les caractères.

`closegraph`
retourne au mode texte

Exemple

```
include "\\prolog\\grapdecl.pro"

goal
   initgraph(detect,0,Graph,Mode,"\\prolog\\bgi"),
   setbkcolor(red), /* ou setbkcolor(4) */
   circle(320,100,100),/* dessine un cercle rouge */
   readchar(_),
   /* attend que l'utilisateur enfonce une touche */
   closegraph. /* retour au mode texte */
```

Remarque :

Attention à la syntaxe un peu particulière de Turbo Prolog, les chemins sont spécifiés avec un double antislash (\\). A l'installation du logiciel, le fichier "grapdecl.pro" est copié dans le répertoire de Prolog et les fichiers .bgi dans le répertoire bgi. Le programme donné ci-dessus spécifie donc les chemins correspondants quelque soit le répertoire actif du disque.

Principales routines graphiques de la version 2.0

Tous les paramètres des routines graphiques (version 1.0 et 2.0) et sonores sont liés et entiers. Il existe une quarantaine de prédicats graphiques, nous n'en donnons ici que quelques uns, pour plus d'informations, consulter le manuel du fabricant.

`bar(x1,y1,x2,y2)`
dessine un rectangle plein , "x1", "y1" et "x2", "y2" désignent les coordonnées supérieur gauche et inférieur droit du rectangle.

`circle(x,y,rayon)`
dessine un cercle de centre "x", "y" et de rayon spécifié?

`cleardevice`
efface l'écran.

`line(x1,y1,x2,y2)`
trace une ligne entre les points spécifiés.

`putpixel(x,y,couleur)`
affiche un point de coordonnées "x", "y" dans la couleur spécifiée.

`rectangle(x1,y1,x2,y2)`
similaire à bar, mais le rectangle est vide.

2) Graphisme de la version 1.0

Prédicats de mise en oeuvre du mode graphique 1.0

`graphics(mode,couleur_avant_plan,couleur_arrière_plan)`
active le mode graphique. le paramètre lié `mode` a la même signification que pour le prédicat `initgraph`.

`text`
Retour au mode texte

Principales routines graphiques de la version 1.0

Il existe deux modes graphiques : le mode normal et le mode tortue du Logo.

Mode normal :

`dot(ligne,colonne,couleur)`
affiche en `ligne` et `colonne` un point de la couleur spécifiée.

`line(ligne1,col1,lig2,col2,couleur)`
trace une ligne entre les points spécifiés et de la couleur spécifiée.

Mode tortue :

`pendown et penup`
abaisse et lève le crayon.

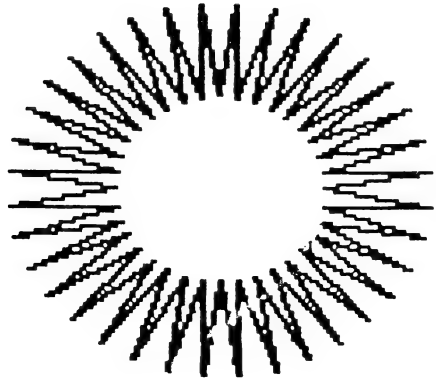
`pencolor(couleur)`
spécifie la couleur du tracé.

`forward(déplacement) et back(déplacement)`
déplacent la tortue vers l'avant ou l'arrière du nombre de pas spécifié (de 1 à 32 000).

`right(angle)` et `left(angle)`
tourne la tortue de `angle` degrés (de 1 à 360) vers la droite où la gauche.

Exemple :

```
predicates
  zigzag(integer)
/*Dessin d'une étoile*/
goal
  graphics(1,1,0),
  zigzag(40).

clauses
  zigzag(0):-
    readchar(_).
  zigzag(N):-
    right(10),
    forward(3000),
    left(20),
    back(3000),
    N1 = N-1,
    zigzag(N1).
```



2.15.3. Son

`beep`
émet un son bref

`sound(durée,fréquence)`
émet un son de la fréquence spécifiée pendant la durée spécifiée.

2.15.4. Accès au DOS

`date(année,mois,jour)` et
`time(heures,minutes,seconges,centièmes)`
lisent ou fixent la date et l'heure de l'horloge interne suivant que les paramètres (entiers) sont libres ou liés.

2.15.5. Fichiers inclus

La directive
`include "nom_fichier_dos"`
inclut un fichier Turbo Prolog durant la compilation du programme principal. Les fichiers doivent être inclus dans un programme là où se situent les mots clés : `domains`, `predicates`, `goal` ou `clauses`.

2.15.6. Interface avec d'autres langages

Bien qu'il soit possible d'effectuer des calculs avec Turbo Prolog, ce dernier n'est pas le mieux adapté au calcul scientifique. C'est pourquoi Turbo Prolog autorise une interface avec Pascal, C, Fortran et l'assembleur. Pour plus d'informations sur l'appel de procédures externes, consulter le manuel du fabriquant.

TROISIÈME PARTIE :

DEUX PROGRAMMES DE BASE POUR RÉALISER UN SYSTEME EXPERT

3.1. UN SYSTEME GENERATEUR DE SYSTEMES EXPERTS

3.1.1. Représentation des connaissances

La réalisation d'un S.G.S.E. implique la création d'une base des connaissances dynamique. Il faut donc concevoir une représentation des règles sous forme de faits et construire un moteur d'inférence capable d'explorer la base. Les connaissances peuvent être représentées par un arbre. Chaque noeud de l'arbre correspond à une règle. La tête d'une règle ou encore son but ou son critère est vraie si la règle père est vraie et si un certain nombre de conditions propres à la règle sont vraies. Au sommet de l'arbre, on trouve le critère de niveau le plus bas. A chaque fois que l'on descend d'un niveau dans l'arbre des connaissances on trouve les règles du niveau inférieur.

3.1.2. Codage des connaissances

Les régles sont rentrées en langage naturel par l'utilisateur, le système expert en assurant le codage interne. Une règle sera représentée par le prédicat :

```
regle(numéro,critère_père,critère,liste_conditions)
```

Où : `numéro`, de type `integer`, spécifie le numéro de la règle. La numérotation des règles est arbitraire, elle est impérative pour pouvoir modifier, supprimer ou afficher une règle et également pour que le système expert puisse expliquer son raisonnement.
`critère`, de type symbol, spécifie la tête de la règle.
`critère_père`, de type `symbol`, spécifie la tête de la règle père.
`liste_conditions`, de type liste d'entiers spécifie la liste des numéros des conditions de la règle.

Une condition sera représentée par le prédicat :

```
cond(numéro,condition)
```

où :

`numéro`, de type `integer` spécifie le numéro de la condition.
`condition`, de type `string`, spécifie une condition.

Exemple :

La règle :
L'animal est un mammifère
SI il a des poils

ET il donne du lait
sera saisie sous la forme suivante :

SGSE :	Donner le critère :
Utilisateur :	mammifère <-┘
SGSE :	Donner le critère père :
Utilisateur :	animal <-┘
SGSE :	Donner la liste des conditions :
Utilisateur :	a des poils <-┘
SGSE :	Condition suivante
Utilisateur :	donne du lait <-┘
SGSE :	Condition suivante
Utilisateur :	<-┘

La règle sera codée par le système expert :

```
regle(1,"animal","mammifère",[2,1]).
cond(1,"donne du lait").
cond(2,"a des poils").
```

3.1.3. Moteur d'inférence

Le moteur d'inférence (réalisé par la clause `exec('3')`), fonctionne en chaînage avant. L'utilisateur donne son critère initial (celui-ci peut être différent du critère racine). Le moteur part alors à la recherche des solutions en explorant la base. Une solution est trouvée lorsqu'une feuille est atteinte. Il y a échec s'il n'y a plus de règles applicables.
Le dialogue est mené par le système expert, l'utilisateur devant répondre aux questions qui lui sont posées par "oui" ou "non". Il n'y a donc aucune compréhension du langage naturel.
Les réponses de l'utilisateur sont mémorisées par le prédicat :

```
propriete(numéro,réponse)
```

où :

`Numéro`, de type `integer`, correspond au numéro de la question.
`réponse`, de type `symbol`, est la réponse de l'utilisateur.

3.1.4. Programme

```
domains liste = integer*
 liste_sym = symbol*

database
 propriete(integer,symbol)
 regle(integer,symbol,symbol,liste)
 cond(integer,symbol)

predicates
 cherche(symbol,liste,liste)
 essai(liste)
 efface1
 efface2
 efface3
 debut
```

```
 exec(char)
 ajout(liste_sym,liste_sym)
 fin_regle(integer,integer)
 fin_cond(integer,integer)
 cherche_cond(liste_sym,liste,liste)
 affiche_cond(liste)
 verif(liste)
 inverse(liste,liste)
 renverse(liste,liste,liste)
 ecrire(liste)
 affiche_regle(integer)
 tester(symbol,symbol)
 sauve(char)

goal
 debut.

clauses
 debut:-
   makewindow(1,7,7,"",0,0,25,80),
   clearwindow,

/*Menu*/
   write("Charger une base (1)"),nl,
   write("Sauver la base (2)"),nl,
   write("Interroger la base (3)"),nl,
   write("Ajouter une règle (4)"),nl,
   write("Lister la base (5)"),nl,
   write("Afficher une règle (6)"),nl,
   write("Supprimer ou modifier une règle (7)"),nl,
   write("Détruire une base (8)"),nl,
   write("Quitter (9)"),nl,
   nl,write("Taper le numéro correspondant à votre choix :"),
   readchar(X),
   clearwindow,
   exec(X),
 debut.
 debut.

/*Charger une base*/
 exec('1'):-
   write("Nom de la base à charger :"),nl,
   effacel,efface2,efface3,
   readln(Nbase),
   existfile(Nbase), consult(Nbase).

 exec('1'):-
   write("Il n'existe pas de base de ce nom"),
   readchar(_).

/*Sauver la base active*/
 exec('2'):-sauve('O').
```

```
/*Interroger la base*/
 exec('3'):-
   effacel,
   nl,write("Donnez votre critère initiale :"),nl,
   readln(C),
   cherche(C,[],L),
   write("Pour parvenir à la solution. "),
   write("J'ai appliqué les règles :"),nl,
   inverse(L,L1),
   ecrire(L1),
   readchar(_).

/*Ajouter de nouvelle règles*/
 exec('4'):-
   clearwindow,
   write("Donner le critère :"),nl,
   readln(C),C<>"",
   write("Donner le critère père :"),nl,
   readln(P),
   write("Donner la liste des conditions "),nl,
   ajout([],L),nl,
   cherche_cond(L,[],L2),
   fin_regle(1,N),
   assertz(regle(N,P,C,L2)),
   exec('4').

/*Lister la base*/
 exec('5'):-clearwindow,
   affiche_regle(_),
   nl,fail.

/*Afficher une règle*/
 exec('6'):-
   write("Numéro de la règle à visualiser :"),nl,
   readint(N),
   clearwindow,
   affiche_regle(N).

/*Supprimer une règle*/
 exec('7'):-
   write("Pour modifier une règle :"),nl,
   write("commencer par la supprimer. "),nl,
   write("Puis appeler le module : Ajouter."),nl,
   readchar(_),
   clearwindow,
   write("Numéro de la règle à supprimer :"),nl,
   readint(N),
   retract(regle(N,_,_,_)).

/*Détruire une base*/
 exec('8'):-
   write("Nom de la base à supprimer : "),nl,
   readln(B),
   deletefile(B).
```

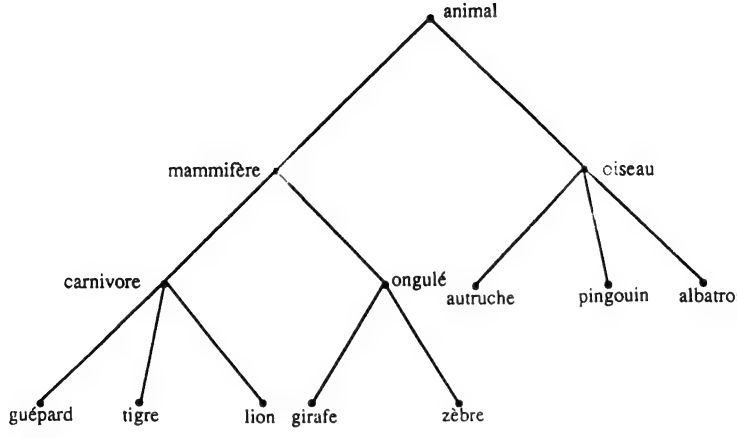
```
/*Quitter le programme*/
 exec('9'):-
   write("Voulez-vous sauver la base (O/N) ?"),
   readchar(X),
   sauve(X),
   !,effacel,efface2,
   efface3,fail.

 exec(_)./*assure la boucle principale*/

/*Sauvegarde de la base*/
 sauve('O'):-
   nl,write("Nom de la base à sauver :"),nl,
   readln(Nbase),
   save(Nbase).

sauve('N').

sauve(_):-
  clearwindow,
  write("Répondez par 'O' ou 'N' "),
  readchar(X),
  sauve(X).

/*Affichage des conditions*/
 affiche_cond([]):-!.
 affiche_cond([T|Q]):-
   cond(T,P),
   write("il ",P),nl,
   not(verif(Q)),
   write("ET "),
   affiche_cond(Q).

/*Détecte la fin des conditions*/
 verif([]):-readchar(_),nl,nl.

/*Transformation de la liste des
conditions symboliques en liste d'entiers*/
 cherche_cond([],L,L):-!.
 cherche_cond([T|Q],L2,L3):-
   cond(N,T),cherche_cond(Q,[N|L2],L3).
 cherche_cond([T|Q],L2,L3):-
   fin_cond(1,N),
   assertz(cond(N,T)),
 cherche_cond(Q,[N|L2],L3).

/*Construire la liste des conditions symboliques*/
 ajout([""|L],L).
 ajout(L2,L3):-readln(X),
   write("Condition suivante :"),nl,
 ajout([X|L2],L3).

/*Ajouter un nouveau numéro de règle*/
 fin_regle(N,N):-not(regle(N,_,_,_)),!.
 fin_regle(N,N1):- H= N+1,
 fin_regle(H,N1).
```

```
/*Ajouter un nouveau numéro de condition*/
 fin_cond(N,N):-not(cond(N,_)),!.
 fin_cond(N,N1):- H= N+1,
 fin_cond(H,N1).

/*Recherche de la solution*/
 cherche(Y,L,L):- not(regle(_,Y,_,_)),
   regle(1,X,_,_),
   write("L'(e) ",X," est un(e) ",Y),
   readchar(_),
   clearwindow.
 cherche(Y,L,L2):-regle(N,Y,X,I),essai(I),
 cherche(X,[N|L],L2). cherche(_,[],[]):-
   write("Je ne trouve pas la solution"),
   readchar(_),fail.

/*Recherche des conditions*/
 essai([]).
 essai([T|Q]):-propriete(T,"oui"),!,essai(Q).
 essai([T|_]):-propriete(T,"non"),!,fail.
 essai([T|Q]):-cond(T,R),
   regle(1,X,_,_),
   write("Est-il vrai que l'(e) ",X," ",R," ?"),nl,
   readln(Rep),tester(Rep,Rep2),
   asserta(propriete(T,Rep2)),
   Rep2="oui",
 essai(Q).

/*Repose la question si la réponse est
différente de oui ou non*/
 tester(Rep,Rep):-Rep="oui",!.
 tester(Rep,Rep):-Rep= "non",!.
 tester(_,Rep):-
   write ("Répondez par oui ou non"),nl,
   readln(Rep),tester(Rep,_).

/*Effacer les réponses de l'utilisateur*/
 efface1:- retract(propriete(_,_)),fail.
 efface1.
/*Effacer la base active*/
 efface2:-retract(regle(_,_,_,_)),fail.
 efface2.
 efface3:-retract(cond(_,_)),fail.
 efface3.

/*inversion de deux listes*/
 inverse(L1,L2):-renverse(L1,[],L2).

 renverse([],L3,L3).
 renverse([T1|Q1],L2,L3):-
 renverse(Q1,[T1|L2],L3).

/*affichage de la liste des règles utilisées*/
 ecrire([]).
 ecrire([T|Q]):-
   write(T," "),ecrire(Q).
```

```
affiche_regle(N):-
regle(N,P,C,L),
  write("Règle numéro : ",N),nl,
  write("L'(e) ",P," est ",C),nl,
  write("SI "),
  affiche_cond(L),
  readchar(_),nl.
```

3.1.5. Exemple d'application

Arbre des connaissances



Liste des règles

```
Règle numéro : 1
L'(e) animal est un(e) mammifère
SI il a des poils
ET il donne du lait

Règle numéro : 2
L'(e) animal est un(e) oiseau
SI il a des plumes
ET il pond des oeufs

Règle numéro : 3
L'(e) mammifère est un(e) carnivore
SI il mange de la viande

Règle numéro : 4
L'(e) mammifère est un(e) ongulé
SI il rumine

Règle numéro : 5
L'(e) ongulé est un(e) girafe
SI il a des taches noires
ET il a un long cou
ET il a de longues pattes
```

```
Règle numéro : 6
L'(e) ongulé est un(e) zèbre
SI il a des rayures noires

Règle numéro : 7
L'(e) oiseau est un(e) autruche
SI il a de longues pattes
ET il a un long cou
ET il est de couleur noire et blanche

Règle numéro : 8
L'(e) oiseau est un(e) pingoin
SI il est de couleur noire et blanche
ET il ne nage pas

Règle numéro : 9
L'(e) oiseau est un(e) albatros
SI il peut voler loin

Régle numéro : 10
L'(e) carnivore est un(e) guépard
SI il est de couleur fauve
ET il a des taches noires

Règle numéro : 11
L'(e)carnivore est un(e) tigre
SI il est de couleur fauve
ET il a des rayures noires

Règle numéro : 12
L'(e) carnivore est un(e) lion
SI il est de couleur fauve
```

Listing de la base des connaissances

animaux

```
regle(1,"animal","mammifère",[2,1])
regle(2,"animal","oiseau",[4,3])
regle(3,"mammifère","carnivore",[5])
regle(4,"mammifère","ongulé",[6])
regle(5,"ongulé","girafe",[9,8,7])
regle(6,"ongulé","zèbre",[10])
regle(7,"oiseau","autruche",[12,8,11])
regle(8,"oiseau","pingoin",[14,13])
regle(9,"oiseau","albatros",[15])
regle(10,"carnivore","guépard",[16,9])
regle(11,"carnivore","tigre",[16,10])
regle(12,"carnivore","lion",[16])

cond(1,"donne du lait")
cond(2,"a des poils")
cond(3,"pond des oeufs")
cond(4,"a des plumes")
cond(5,"mange de la viande")
cond(6,"rumine")
cond(7,"a delongues pattes")
```

```
cond(8,"a un long cou")
cond(9,"a des taches noires")
cond(10,"a des rayures noires")
cond(11,"est de couleur noire")
cond(12,"a de longues pattes")
cond(13,"ne nage pas")
cond(14,"est de couleur noire et blanche")
cond(15,"peut voler loin")
cond(16,"est de couleur fauve")
```

Exemple d'interrogation

Après avoir chargé la base des connaissances, sélectionner le module "Interroger la base (3)", le programme affichera :

```
Donnez votre critère initial :
animal
Est-il vrai que l'animal a des poils ?
oui
Est-il vrai que l'animal donne du lait ?
oui
Est-il vrai que l'animal mange de la viande ?
oui
Est-il vrai que l'animal est de couleur fauve ?
oui
Est-il vrai que l'animal a des taches noires ?
oui
L'animal est un(e) guépard.
Pour parvenir à la solution.
J'ai appliqué les règles :
1, 3, 10
```

3.2. UN PETIT ANALYSEUR SYNTAXIQUE

3.2.1. Position du problème

Le programme analyse une phrase de la forme :
`groupe_sujet + verbe + groupe_complément`

Un groupe_sujet est : `un déterminant + un nom`
Un verbe est : `un verbe`

Un groupe_complément est : `un déterminant + un nom`
Un déterminant est : `rien OU un déterminant`

Le programme lit une phrase en entrée, vérifie si les mots appartiennent au dictionnaire et si la syntaxe de la phrase correspond à la grammaire. Dans ce cas, le programme affiche la structure de la phrase. Le programme est rudimentaire, ne vérifie pas la concordance du genre ni de la personne. Le dictionnaire est inclus dans le programme. Il est bien évident qu'un véritable analyseur syntaxique exigerait une base de faits (un dictionnaire) dynamique.
Le programme commence par transformer la phrase en une liste de symboles, puis parcourt cette liste en vérifiant que chaque mot appartient au dictionnaire et construit simultanément

une liste numérique correspondant à la nature de chaque mot. La liste numérique ainsi construite est unifiée aux diverses formes de phrases possibles et la structure affichée.

3.2.2. Programme

```
domains
 phrase =
phrase(groupe_sujet,verbe,groupe_complément)
 groupe_sujet = groupe_sujet(det,nom)
 verbe = verbe(symbol)
 groupe_complément = groupe_complément(det,nom)
 det= rien;det(symbol)
 nom = nom(symbol)
 liste = symbol*
 liste_ent = integer*

predicates
 deter(symbol)
 verbes(symbol)
 nom(symbol)
 mot(symbol,integer)
 est_nom(symbol)
 est_verbe(symbol)
 egal_phrase(phrase,phrase)
 analyse(liste,liste_ent,liste_ent)
 liste_chaine(string,liste)
 inverse(liste_ent,liste_ent)
 renverse(liste_ent,liste_ent,liste_ent)
 debut
 egal(liste_ent,liste)

goal
 debut.

clauses
/*Base de faits : dictionnaire*/
 nom(chat).
 nom(souris).
 nom("Pierre").
 nom("Marie").
 nom(pomme).
 deter(le).
 deter(la).
 deter(les).
 deter(un).
 deter(une).
 deter(des).
 verbes(mange).
 verbes(aime).
 verbes(attrape).
```

```
debut:-
  makewindow(1,7,7,"",0,0,25,80),
  write("Donnez la phrase à analyser :"),nl,
  readln(X),liste_chaine(X,Y),
  analyse(Y,[],L),
  inverse(L,Liste),
  egal(Liste,Y).

/*Retourne une liste de symboles correspondant à la phrase*/
liste_chaine(C,[T|Q]):-
  fronttoken(C,T,X),!,liste_chaine(X,Q).
liste_chaine(_,[]).

egal_phrase(P,P).

/*Retourne une liste d'entiers correspondant aux
mots de la phrase*/
analyse([],L,L).
analyse([T1|Q1],L2,L3):-
  mot(T1,Y),analyse(Q1,[Y|L2],L3).

/*Retourne la phrase analysée*/
egal([2,3,2],[X,Y,Z]):-
egal_phrase(P,phrase(groupe_sujet(rien,nom(X)),
  verbe(Y),groupe_complément(rien,nom(Z)))),
  write(P).
egal([1,2,3,1,2],[X,Y,Z,T,U]):-
  egal_phrase(P,phrase(groupe_sujet(det(X),nom(Y)),
  verbe(Z),groupe_complément(det(T),nom(U)))),
  write(P).
egal([1,2,3,2],[X,Y,Z,T]):-
  egal_phrase(P,phrase(groupe_sujet(det(X),nom(Y)),
  verbe(Z),groupe_complément(rien,nom(T)))),
  write(P).

egal([2,3,1,2],[X,Y,Z,T]):-
  egal_phrase(P,phrase(groupe_sujet(rien,nom(X)),
  verbe(Y),groupe_complément(det(Z),nom(T)))),
  write(P).

egal(_,_):-write("Phrase non analysable").

/*inversion d'une liste*/
inverse(L1,L3):-renverse(L1,[],L3).

renverse([],L3,L3).
renverse([T1|Q1],L2,L3):-
  renverse(Q1,[T1|L2],L3).
```

```
/*Reconnaissance des mots de la phrase*/
 mot(X,Y):-deter(X),Y=1,!.
 mot(X,Y):-est_nom(X),Y=2,!.
 mot(X,Y):-est_verbe(X),Y=3,!.
 mot(X,_):-write(X," mot inconnu"),fail.

/*Recherche des pluriels*/
 est_nom(X):-nom(X).
 est_nom(X):-nom(Y),concat(Y,"s",X).
 est_verbe(X):-verbes(X).
 est_verbe(X):-verbes(Y),concat(Y,"nt",X).
```

3.2.3. Exemples d'application

Phrase :
le chat mange la souris
Analyse :

```
phrase(groupe_sujet(det("le"),nom("chat")),
verbe("mange"),groupe_complément(det("la"),nom("souris")))
```

Phrase :
Pierre aime Marie
Analyse :

```
phrase(groupe_sujet(rien,nom("Pierre")),
verbe("aime"),groupe_complément(rien,nom("Marie"))
```

Phrase :
Pierre aime
Analyse :

```
Phrase non analysable
```

Phrase :
Le chien attrape le chat
Analyse :

```
chien mot inconnu
```

Index